

Parallel Spherical Harmonic Transforms on heterogeneous architectures (GPUs/multi-core CPUs)

Mikolaj Szydlarski^{1*}, Pierre Esterie², Joel Falcou², Laura Grigori^{3,4*} and Radek Stompor^{4*}

¹ INRIA Saclay-Île de France, F-91893 Orsay, France

² Université Paris Sud, F-91405 Orsay, France

³ INRIA Rocquencourt, Alpines, B.P. 105, F-78153, Le Chesnay Cedex, France

⁴ UPMC Univ Paris 06, CNRS UMR 7598, Laboratoire Jacques-Louis Lions, F-75005, Paris, France

⁵ APC, Univ Paris Diderot, CNRS/IN2P3, CEA/Irfu, Obs de Paris, Sorbonne Paris Cité, France

SUMMARY

Spherical Harmonic Transforms (SHT) are at the heart of many scientific and practical applications ranging from climate modelling to cosmological observations. In many of these areas new, cutting-edge science goals have been recently proposed requiring simulations and analyses of experimental or observational data at very high resolutions and of unprecedented volumes. Both these aspects pose formidable challenge for the currently existing implementations of the transforms.

This paper describes parallel algorithms for computing SHT with two variants of intra-node parallelism appropriate for novel supercomputer architectures, multi-core processors and Graphic Processing Units (GPU). It also discusses their performance, alone and embedded within a top-level, MPI-based parallelisation layer ported from the S²HAT library, in terms of their accuracy, overall efficiency and scalability. We show that our inverse SHT run on GeForce 400 Series GPUs equipped with latest CUDA architecture ("Fermi") outperforms the state of the art implementation for a multi-core processor executed on a current Intel Core i7-2600K. Furthermore, we show that an MPI/CUDA version of the inverse transform run on a cluster of 128 Nvidia Tesla S1070 is as much as 3 times faster than the hybrid MPI/OpenMP version executed on the same number of quad-core processors Intel Nehalem for problem sizes motivated by our target applications. Performance of the direct transforms is however found to be at the best comparable in these cases. We discuss in detail the algorithmic solutions devised for the major steps involved in the transforms calculation, emphasising those with a major impact on their overall performance, and elucidates the sources of the dichotomy between the direct and the inverse operations.

Copyright © 2012 John Wiley & Sons, Ltd.

Received ...

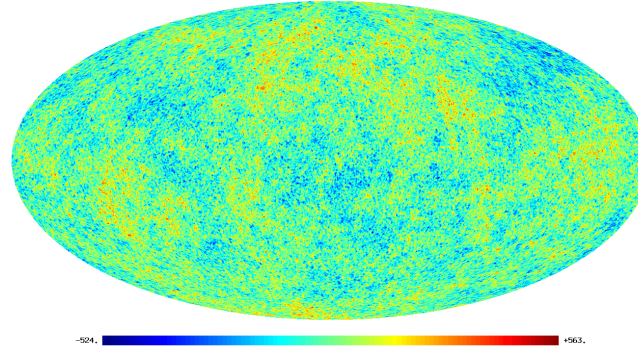
KEY WORDS: Spherical Harmonic Transforms, hybrid architectures, hybrid programming, CUDA, multi-GPU, CMB

1. INTRODUCTION

Spherical harmonic functions constitute an orthonormal complete basis for signals defined on a 2-dimensional sphere. Spherical harmonic transforms are therefore common in all scientific applications where such signals are encountered. These include a number of diverse areas ranging from weather forecasts and climate modelling, through geophysics and planetology, to various applications in astrophysics and cosmology. In these contexts a direct Spherical Harmonic Transform (SHT) is used to calculate harmonic domain representations of the signals, which often possess simpler properties and are therefore more amenable to further investigation. An inverse

*Correspondence to: mikolaj.szydlarski@{inria.fr, gmail.com}, laura.grigori@inria.fr, radek@apc.univ-paris-diderot.fr

SHT is then used to synthesize a sky image given its harmonic representation. Both of those are also used as means of facilitating the multiplication of huge matrices, defined on the sphere, and having the property of being diagonal in the harmonic domain. Such matrices play an important role in statistical considerations of signals, which are statistically isotropic and such operations are among key operations involved in Monte Carlo Markov Chain sampling approaches used to study such signals, e.g. [1].



successfully address all these needs would not only have to scale well in the range of interest, but also be sufficiently quick to be used as part of massive Monte Carlo simulations or extensive sampling solvers, which are both important steps of the scientific exploitation of the CMB data.

At this time there are a few software packages available which permit calculations of the SHT. These include HEALPIX [2], GLESP [3], CCSHT [4], LIBPSHT [5], and S²HAT [6], which are commonly used in the CMB research, and some others such as, SPHARMONICKIT/S2KIT [7] and SPHERPACK [8]. They propose different levels of parallelism and optimization, while implementing, with the exception of two last ones, essentially the same algorithm. Among those, the LIBPSHT package [9] offers typically the best performance, due to efficient code optimizations, and it is considered as the state of the art implementation on serial and shared-memory platforms, at least for some of the popular sky pixelization schemes. Conversely, the implementation offered by the S²HAT library is best adapted to distributed memory machines, and fully parallelized and scalable with respect to both memory usage and calculation time. Moreover, it provides a convenient and flexible framework straightforwardly extensible to allow for multiple parallelization levels based on hybrid programming models. Given the specific applications considered in this work, the distributed memory parallelism seems inevitable, and it is therefore the S²HAT package that we select as a starting point for this work.

The basic algorithm we use hereafter, and which is implemented in both LIBPSHT and S²HAT libraries and described in the next section, scales as $\mathcal{O}(\mathcal{R}_N \ell_{max}^2)$, where \mathcal{R}_N is the number of isolatitudinal rings of pixels in the analyzed map and ℓ_{max} fixes the resolution and thus determines the number of modes in the harmonic domain, equal to $\simeq \ell_{max}^2$. For full sky maps we have usually $\ell_{max} \propto \mathcal{R}_N \propto n_{pix}^{1/2}$, where n_{pix} is the total number of sky pixels, and therefore the typical complexity is $\mathcal{O}(\ell_{max}^3) \propto \mathcal{O}(n_{pix}^{3/2})$. We note that further improvements of this overall scaling are possible. For instance, if two identical transforms have to, or can, be done simultaneously, we could store precomputed data and speed up the computation by a factor of up to 2. However, in practice such speedups are difficult to obtain due to very expensive operations of reading or moving the precomputed data between different hierarchies of memory. In this paper we focus only on the core algorithm and leave an investigation of such potential extensions as future work.

We note that alternative algorithms have been also proposed, some of which display superior complexity. In particular, Driscoll and Healy [10] proposed a divide-and-conquer algorithm with a theoretical scaling of $\mathcal{O}(n_{pix} \ln^2 n_{pix})$. This approach is limited to special equidistant sphere grids and being inherently numerically unstable, it requires corrective measures to ensure high precision results. This in turn affects its overall performance. For instance, the software SPHARMONICKIT/S2KIT, which is the most widely used implementation of this approach, has been found a factor 3 slower than the HEALPIX transforms implementing the standard $\mathcal{O}(n_{pix}^{3/2})$ method at the intermediate resolution of $\ell_{max} \sim 1024$ [11]. Other algorithms also exist and typically involve a precomputation step of the same complexity, but often less favourable prefactors, as the standard approach, and an actual calculation of the transforms, which typically exploits either the Fast Multipole Methods, e.g., [12, 13] or matrix compression techniques, e.g., [14, 15], to bring down its overall scaling to $\mathcal{O}(n_{pix} \ln n_{pix})$ [12], $\mathcal{O}(n_{pix} \ln^2 n_{pix})$ [13, 15], or $\mathcal{O}(n_{pix}^{5/4} \ln n_{pix})$ [14]. The methods of this class typically require significant memory resources needed to store the precomputation products and are advantageous only if a sufficient number of successive transforms of the same type has to be performed in order to compensate for the precomputation costs. The most recent, and arguably satisfactory, implementation of such ideas is a package called WAVEMOTH[¶] [16], which achieves a speed-up of the inverse SHT with respect to the LIBPSHT library by a factor ranging from 3 to 6 for $\ell_{max} \sim 4000$. In such a case the required extra memory is on the order of 40 GBytes and it depends strongly, i.e., $\propto \ell_{max}^3$, on the resolution. We also note that in some applications the need to use SHT can be sometimes by-passed by resorting to approximate but numerically efficient means such as the Fast Fourier Transforms as for instance in the context of convolutions on the sphere as discussed in [17].

[¶]WAVEMOTH: <https://github.com/wavemoth/wavemoth>

In many practical applications, as the ones driving this research, SHT are just one of many processing steps, which need to be performed to accomplish a final task. In such a context, the memory consumption and the ability of using large number of processors frequently emerge as the two most relevant requirements, as the cost of the SHT transforms is often subdominant and their complexity less important than that of the other operations. From the memory point of view, the standard algorithm has the smallest memory footprint and therefore remains an attractive option. The important question is then whether its efficient, scalable, parallel implementation is indeed possible. Such a question is particularly pertinent in the context of heterogeneous architectures and hybrid programming models and this is the context investigated in this work.

The past work on the parallelisation of SHT includes a parallel version of the two step algorithm of Driscoll and Healy introduced by [18], the algorithm of [19] based on rephrasing the transforms as matrix operations and therefore making them effectively vectorizable, and a shared memory implementation available in the LIBSHT package. More recently, algorithms have been developed for GPUs [20, 21]. This latter work provided a direct motivation for the investigation presented here, which describes what, to the best of our knowledge, is the first hybrid design of parallel SHT algorithms suitable for a cluster of GPU-based architectures and current high-performance multi-core processors, and involving hybrid OpenMP/MPI and MPI/CUDA programming. We find that once carefully optimised and implemented, the algorithm displays nearly perfect scaling in both cases, extending at least up to 128 MPI processes mapped on the same number of pairs of multi-core CPU-GPU. We also find that inverse SHT run on GeForce 400 Series GPGPUs equipped with the latest CUDA device ("Fermi") outperforms a state of the art implementation for multi-core processors executed on latest Intel Core i7-2600K, while the direct transforms in both these cases perform comparably.

This paper is organised as follows. In section 2 we introduce the algebraic background of the spherical harmonic transforms. In section 3 we describe a basic, sequential algorithm and list useful assumptions concerning the pixelisation of a sphere, which facilitate a number of acceleration techniques used in our approach. In the following section we introduce a detailed description of our parallel algorithms along with two variants suitable for clusters of GPUs and clusters of multi-cores. Section 5 presents results for both implementations and finally section 6 concludes the paper.

2. ALGEBRAIC BACKGROUND

2.1. Definitions and notations

Spherical harmonic functions constitute a complete, orthogonal basis on the sphere. They can be therefore used to define a unique decomposition of any function defined on the sphere, $f : S^2 \rightarrow \mathbb{R}$, and depending on two spherical coordinates: $\theta \in (0, \pi)$ (colatitude) and $\phi \in [0, 2\pi)$ (longitude – measured counterclockwise about the positive z -axis from the positive x -axis),

$$f(\theta, \phi) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \mathbf{a}_{\ell m} Y_{\ell m}(\theta, \phi), \quad (1)$$

where $\mathbf{a}_{\ell m}$ are spherical harmonic coefficients representing f in the harmonic domain, $Y_{\ell m}$ is a spherical harmonic basis function of degree ℓ and order m . The spherical harmonic coefficients can be then computed by taking a scalar inner product of f with the corresponding spherical harmonic, which can be expressed as a 2-dimensional integral

$$\mathbf{a}_{\ell m} = \int_0^{\pi} d\theta \int_0^{2\pi} d\phi f(\theta, \phi) Y_{\ell m}^{\dagger}(\theta, \phi) \sin \theta, \quad (2)$$

where \dagger denotes complex conjugation.

In actual applications, the functions defined on the sphere are typically discretised and given only on a set of grid points. In this case, the spherical harmonic transforms are used either to project the

data, s_n , onto the harmonic modes – an operation referred to as an *analysis* step or a direct transform – or to reconstruct the discretised data given a set of harmonic coefficients, $a_{\ell m}$ – called an inverse transform or a *synthesis* step. These steps can be written as,

$$\tilde{a}_{\ell m} = \sum_{\{\theta_n, \phi_n\}} w_n s_n(\theta_n, \phi_n) Y_{\ell m}(\theta_n, \phi_n), \quad (3)$$

$$\tilde{s}_n(\theta_n, \phi_n) = \sum_{\ell=0}^{\ell_{max}} \sum_{m=-\ell}^{\ell} a_{\ell m} Y_{\ell m}(\theta_n, \phi_n). \quad (4)$$

Given the choice of a bandwidth, ℓ_{max} , pixels weights, w_n , and the grid geometry, both these transforms may only be approximate, what is indicated by a tilde over the computed quantities. This is because of the sky sampling issues on the analysis step, or the assumed bandwidth in the case of the synthesis.

In CMB applications, s_n is typically a vector of real-valued, pixelized data, e.g. the brightness of incoming CMB photons, assigned to n_{pix} locations, (θ_n, ϕ_n) , on the sky corresponding to centres of suitable chosen sky pixels, and is referred to hereafter as *a map*. The parameter ℓ_{max} defines the maximum order of the Legendre function and thus a band-limit of the field s_n , and in practice it is set by the experimental resolution.

The basis functions $Y_{\ell m}(\theta_n, \phi_n)$ are defined in terms of normalised associated Legendre functions, $\mathcal{P}_{\ell m}$, of degree ℓ and order m ,

$$Y_{\ell m}(\theta_n, \phi_n) \equiv \mathcal{P}_{\ell m}(\cos \theta_n) e^{im\phi_n}. \quad (5)$$

As the associated Legendre functions are real-valued, we have that $Y_{\ell -m} = (-1)^m Y_{\ell m}^\dagger$, which we will use later on to compute all spherical harmonics in terms of the associated Legendre functions $\mathcal{P}_{\ell m}$ with $m \geq 0$. We note that from this we also have that $a_{\ell m}^\dagger = (-1)^m a_{\ell -m}$ for any real-valued data, s .

Using equation (5) we can separate the variables on the rhs of equations (3) and (4) thus reducing the computation of the spherical harmonic transform to a Fourier transform in the longitudinal coordinate ϕ_n followed by a projection onto the associated Legendre functions,

$$\tilde{a}_{\ell m} = \sum_{\{\theta_n\}} \left[\left(\sum_{\{\phi_n\}} w_n s_n(\theta_n, \phi_n) e^{im\phi_n} \right) \mathcal{P}_{\ell m}(\cos \theta_n) \right]. \quad (6)$$

The associated Legendre functions satisfy the following recurrence (with respect to the multipole number ℓ for a fixed value of m) critical for the algorithms developed in this paper,

$$\mathcal{P}_{\ell+2,m}(x) = \beta_{\ell+2,m} x \mathcal{P}_{\ell+1,m}(x) + \frac{\beta_{\ell+2,m}}{\beta_{\ell+1,m}} \mathcal{P}_{\ell m}(x), \quad (7)$$

where

$$\beta_{\ell m} = \sqrt{\frac{4\ell^2 - 1}{\ell^2 - m^2}}. \quad (8)$$

The recurrence is initialised by the starting values,

$$\begin{aligned} \mathcal{P}_{mm}(x) &= \frac{1}{2^m m!} \sqrt{\frac{(2m+1)!}{4\pi}} (1-x^2)^m \\ &\equiv \mu_m (1-x^2)^m, \end{aligned} \quad (9)$$

$$\mathcal{P}_{m+1,m}(x) = \beta_{\ell+1,m} x \mathcal{P}_{mm}(x). \quad (10)$$

The recurrence is numerically stable but a special care has to be taken to avoid under- or overflow for large values of ℓ_{max} [22] e.g., for increasing m the \mathcal{P}_{mm} values can become extremely small such

that they can no longer be represented by the binary floating-point numbers in IEEE 754^{||} standard. However, since we have freedom to rescale the values of $\mathcal{P}_{\ell m}$, we can use it to avoid the problem. Consequently, on each step of the recurrence, the newly computed value of the associated Legendre function is tested and rescaled together with the preceding value, if found to be too close to the over- or underflow limits. The rescaling coefficients are kept track of, e.g., accumulated in form of their logarithms, and used to scale back all the computed values of $\mathcal{P}_{\ell m}$ at the end of the calculation. This scheme is based on two facts. First, the values of the associated Legendre functions calculated via the recurrence change gradually and rather slowly on each step. Second, their actual values as needed by the transforms are representable within the range of the double precision values.

The specific implementation of these ideas used in all our algorithms follows that of the S²HAT software and the HEALPix package (at least up to its version 2.0). It employs a precomputed vector of values, sampling the dynamic range of the representable double precision numbers and thus avoids any explicit computation of numerically-expensive functions, such as logarithms and exponentials. This scaling vector is used to compare the values of $\mathcal{P}_{\ell m}$ computed on each step of the recurrence, and then it is used to rescale them if needed. For sake of simplicity, in this paper we assume that this is an integral part of associated Legendre function evaluation (independent of the platform architecture) and we omit an explicit description of the algorithm used for rescaling. For more details we refer to [21] and the documentation of the HEALPix package [2].

2.2. Specialised formulation

For a general grid of n_{pix} points on the sphere, a direct calculation of any of the SHT has computational complexity $\mathcal{O}(\ell_{max}^2 n_{pix})$, where the factor $\mathcal{O}(\ell_{max})$ reflects the cost of calculating all necessary associated Legendre functions up to the order ℓ_{max} for each of n_{pix} points on the sphere. This is clearly prohibitive for problems in the range of our interest. For instance, the current and forthcoming balloon-borne and ground-based experiments will produce maps of sky containing as many as $n_{pix} \in [10^5, 10^6]$ pixels and up to $\mathcal{O}(10^6 - 10^7)$ harmonic modes. Maps from the already operating Planck satellite will consist of between $\mathcal{O}(10^6)$ and $\mathcal{O}(10^8)$ pixels and harmonic modes. To improve on such complexity we will make some additional geometrical constraints, which are to be imposed on any permissible pixelization, and which ensure that the numerical cost is down to $\mathcal{O}(n_{pix}^{1/2} \ell_{max}^2)$. We note that nearly all spherical grids, which are currently used in CMB analysis (for instance HEALPix and GLESP), as well as in many other application areas, conform with these assumptions, which are [22],

- the map pixels are arranged on $\mathcal{R}_N \approx \sqrt{n_{pix}}$ iso-latitudinal rings, where latitude of each ring is identified by a unique polar angle θ_n (for sake of simplicity hereafter we will refer to this angle by the ring index i.e., $r_n \rightarrow \theta_n$, where $n \in [0, \mathcal{R}_N]$)
- within each ring r_n , pixels must be equidistant ($\Delta\phi = \text{CONST}$), and therefore, $\phi_n = \phi_0 + n \Delta\phi$, though their number can vary from a ring to another ring.

The requirement of the iso-latitude distribution for all pixels helps to cut on the number of required floating point operations (FLOPs) as the associated Legendre function need now be calculated only once for each pixel ring.

Taking into account these restrictions and the definition of the basis function, $Y_{\ell m}$ equation (5), we can rewrite (adapted from [22]) the synthesis step from equation (4) as

$$s(r_n, \phi_n) = \sum_{m=-\ell_{max}}^{\ell_{max}} e^{im\phi_0} \Delta_m^A(r_n), \quad (11)$$

^{||}The IEEE has standardised the computer representation for binary floating-point numbers in IEEE 754. This standard is followed by almost all modern machines.

where ϕ_0 denotes the ϕ angle of the first pixel in the ring and $\Delta_m^A(r_n)$ denotes a set of functions defined as,

$$\Delta_m^A(r_n) \equiv \begin{cases} \sum_{\ell=0}^{\ell_{max}} a_{\ell 0} \mathcal{P}_{\ell 0}(\cos r_n), & m = 0, \\ \sum_{\ell=m}^{\ell_{max}} a_{\ell m} \mathcal{P}_{\ell m}(\cos r_n), & m > 0, \\ \sum_{\ell=|m|}^{\ell_{max}} a_{\ell|m|}^\dagger \mathcal{P}_{\ell|m|}(\cos r_n), & m < 0. \end{cases} \quad (12)$$

And similarly for the analysis step, equation (3),

$$a_{\ell m} = \sum_{n=0}^{\mathcal{R}_N} \Delta_m^S(r_n) \mathcal{P}_{\ell m}(\cos r_n), \quad (13)$$

where

$$\Delta_m^S(r_n) \equiv \sum_{\{\phi_n\}} w s(r_n, \phi_n) e^{-im\phi_0}. \quad (14)$$

From above, we can see that the spherical harmonic transforms can be split into two successive computations: one calculating the associated Legendre functions, equations (12) and (13), and the other performing the series of Fast Fourier Transforms (FFTs), equations (11) and (14) [23]. This splitting immensely facilitates the implementation. For instance, in order to evaluate Fourier transforms we can use third-party libraries addressing this problem. In our application, the number of samples (pixels) per ring may vary and its prime decomposition may involve large prime factors. A care has to be therefore taken in selecting an appropriate FFT library. For instance, the well-known, standard version of the `fftpack` library [24] performs well only for data vectors of the length N_{FFT} , whose prime decomposition contains the factors 2, 3, and 5 and its numerical complexity degrades from $\mathcal{O}(N_{FFT} \log N_{FFT})$ to $\mathcal{O}(N_{FFT}^2)$ in other cases. HEALPIX and LIBPSHT use therefore its modified version implementing Bluestein's algorithm for FFTs with large prime factors, guaranteeing $\mathcal{O}(N_{FFT} \log N_{FFT})$ complexity. In this work, we employ by default in all versions of our routines the `FFTW` library [25], which ensures similar scaling, competitive performance and portability.

2.3. Time consumption

Three parameters typically determine the data sizes and the complexity of our problem. These are the total number of pixels, n_{pix} , the number of rings of pixels in the analysed map, \mathcal{R}_N , and the maximum order of the associated Legendre functions, ℓ_{max} . In particular the number of modes in the harmonic domain is then $\sim \ell_{max}^2$. In well defined cases these three parameters are not completely independent. For example, in the case of full sky maps, we have usually $\ell_{max} \propto \mathcal{R}_N \propto n_{pix}^{1/2}$. This implies that the recurrence step requires $\mathcal{O}(\mathcal{R}_N \ell_{max}^2)$ FLOPs, due to the fact that for each of the \mathcal{R}_N rings we have to calculate a full set, up to ℓ_{max} , of the associated Legendre functions, $\mathcal{P}_{\ell m}$, at cost $\propto \ell_{max}^2$. The Fourier transform step, as mentioned above, has then complexity $\mathcal{O}(\mathcal{R}_N \ell_{max} \log \ell_{max})$, which should therefore be subdominant as compared to the Legendre transforms. This is indeed confirmed by our experimental results shown in Figure 2 depicting a typical breakdown of the average overall time into the main steps of the SHT computation performed on the quad-core Intel i7-2600K processor. The evaluation of the Legendre transform takes nearly 10 times longer than performing the FFTs. This motivated us to study the possible improvements of this part of the calculation. We explore this issue in the context of the heterogeneous architectures and hybrid programming models. In particular we introduce a parallel algorithm that is suitable for clusters of accelerators, namely multi-core processors and GPUs which we employ for an efficient evaluation of spherical harmonic transforms.

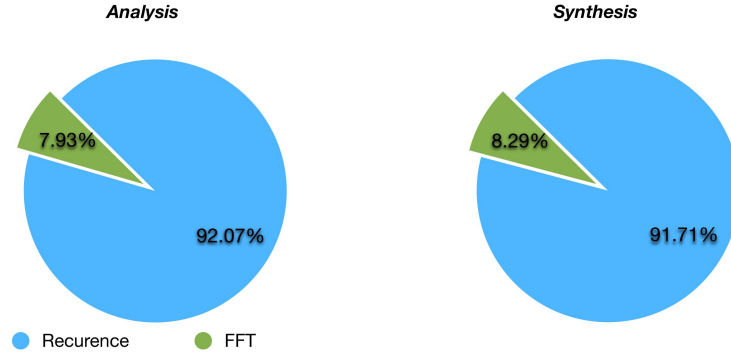


Figure 2. The overall time breakdown between two main steps of the SHT algorithm as computed assuming the same band limit ($\ell_{max} = 4096$) on a quad-core Intel i7-2600K processor.

3. BASIC ALGORITHM

In this section we present basic algorithms for the computation of the spherical harmonic transforms, for sphere pixelizations which conform with the assumptions listed in the previous section. In section 4 we then introduce their parallel versions and discuss possible improvements and amendments as driven by the specific architectures considered in this work.

Algorithm 1 presents a computation of the discrete, direct SHT, and thus implementing equation (3), which for a given bandwidth, ℓ_{max} , and an input data vector, s , computes a corresponding set of the harmonic coefficients $a_{\ell m}$. Algorithm 2 describes a computation of the

Algorithm 1 DIRECT TRANSFORM (EQUATION (3))

Require: $s \in \mathbb{R}_{n_{pix}}$ values

STEP 1 - $\Delta_m^S(r)$ CALCULATION

for every ring r do

for every $m = 0, \dots, m_{max}$ do

 ◦ calculate $\Delta_m^S(r)$ via FFT equation (14)

end for(m)

end for(r)

STEP 2 - PRE-COMPUTATION

for every $m = 0, \dots, m_{max}$ do

 ◦ μ_m pre-computation equation (9)

end for (m)

STEP 3 - CORE CALCULATION

for every ring r do

for every $m = 0, \dots, m_{max}$ do

for every $\ell = m, \dots, \ell_{max}$ do

 ◦ compute $\mathcal{P}_{\ell m}$ via equation (7)

 ◦ update $a_{\ell m}$, equation (13)

end for (ℓ)

end for (m)

end for (r)

return $a_{\ell m} \in \mathbb{C}^{\ell \times m}$

discrete, inverse transform as in equation (4), which takes as input a set of harmonic coefficients, $a_{\ell m}$, and returns a set of real-valued numbers, s .

Algorithm 2 INVERSE TRANSFORM (EQUATION (4))

Require: $\alpha_{\ell m} \in \mathbb{C}^{\ell \times m}$ values

STEP 1 - PRE-COMPUTATION

for every $m = 1, \dots, m_{max}$ **do**

μ_m precomputation equation (9)

end for (m)

STEP 2 - Δ_m^A CALCULATION

for every ring r **do**

for every $m = 0, \dots, m_{max}$ **do**

for every $\ell = m, \dots, \ell_{max}$ **do**

 compute $\mathcal{P}_{\ell m}$ via equation (7)

 update $\Delta_m^A(r)$, equation (14)

end for (ℓ)

end for (m)

end for (r)

STEP 3 - s CALCULATION

for every ring r **do**

for every $m = 0, \dots, m_{max}$ **do**

 calculate s via FFT and given $\Delta_m^A(r)$, equation (11)

end for (m)

end for (r)

return $s \in \mathbb{R}_{n_{pix}}$

3.1. Similarities

The algorithms are almost the same in terms of the number and the type of arithmetic operations involved in their execution (see for instance Figure 2 or a detailed analysis in [26]), employ a similar basic algorithmic principle and have similar structure. Indeed both these algorithms use a divide and conquer approach, as they subdivide the initial problem, dealing with the full set of pixels, into smaller subproblems of a similar form and each involving only the rings of pixels. Each of these subproblems requires the calculation of the associated Legendre functions, which is again solved by subdividing it into as many as ℓ_{max} recurrences with respect to the multipole number ℓ for a fixed value of m and only finally all the intermediate results are combined together to provide the solution of the original problem. Both these algorithms are also similar from the structural point of view as they consist of three main steps:

- pre-computation of the starting values of the recurrence (9) in the loop over all $m \in [0, m_{max}]$ (STEP 2 in Algorithm 1 and STEP 1 in Algorithm 2),
- evaluation of the associated Legendre functions for each ring of pixels via the recurrence from equation (7). This requires three nested loops, where the loop over ℓ is set as the innermost to facilitate the evaluation of the recurrence with respect to the multipole number ℓ for a fixed value of m (STEP 3 in Algorithm 1 and STEP 2 in Algorithm 2).
- calculation of $\Delta_m^{A/S}(\theta)$ using FFTs (STEP 1 in Algorithm 1 and STEP 3 in Algorithm 2).

3.2. Differences

The main difference is related to the dimension of the partial results which need to be updated within the nested loop, where the associated Legendre functions are evaluated (STEP 3 in Algorithm 1 and STEP 2 in Algorithm 2). In the case of the direct transform, on every pass of the loop over rings we need to update all the (non-zero) elements of the matrix $\alpha_{\ell m}$ by the result of the product between the object, Δ_m^S , equation (14), which is precomputed earlier, and the vector of the associated Legendre function with the azimuthal number equal to m . In the case of the inverse transform, the nested loop (STEP 2 in Algorithm 2) is used to compute Δ_m^A , equation (12), that can be expressed as the

matrix-vector product,

$$\Delta_m^A = \begin{pmatrix} \Delta_m^A(r_0, m) \\ \Delta_m^A(r_1, m) \\ \vdots \\ \Delta_m^A(\mathcal{R}_N, m) \end{pmatrix} = \begin{pmatrix} \mathcal{P}_{mm}(\cos r_0) & \dots & \mathcal{P}_{\ell_{max}m}(\cos r_0) \\ \mathcal{P}_{mm}(\cos r_1) & \dots & \mathcal{P}_{\ell_{max}m}(\cos r_1) \\ \vdots & \vdots & \vdots \\ \mathcal{P}_{mm}(\cos r_{\mathcal{R}_N}) & \dots & \mathcal{P}_{\ell_{max}m}(\cos r_{\mathcal{R}_N}) \end{pmatrix} \begin{pmatrix} a_{mm} \\ a_{m+1m} \\ \vdots \\ a_{\ell_{max}m} \end{pmatrix}. \quad (15)$$

Depending on the specific computer architecture, this difference may have a significant effect on the performance of these algorithms in terms of runtime (see further investigation in this paper). We also note that this asymmetry is due to using the recurrence with respect to ℓ for the calculation of the associated Legendre functions, which implies that the loop over ℓ should be placed as innermost, what in turn prevents expressing the calculation of $a_{\ell m}$ in Algorithm 1 as a matrix-vector operation.

4. PARALLEL SPHERICAL HARMONIC TRANSFORMS

4.1. Top-level distributed-memory parallel framework

The need for a distributed memory parallelisation layer is driven by the volumes of the current and anticipated data, both the maps and the harmonic coefficients, and which have to be analysed in the applications targeted here. It is also driven by the required flexibility of the transform implementations. These are envisaged to be frequently needed in massively parallel applications, where SHT typically constitute a single step of an entire processing chain. This requirement calls not only for a flexible interface and an ability to use efficiently as many MPI processes as available, but also for a low memory footprint per process.

The top level parallelism employed in this work is adopted from the S²HAT library originally developed by the last author. Though not new, it has not been described in detail in the literature yet and we present below its first detailed description including its performance model and analysis. The framework is based on a distributed-memory programming model and involves two computational stages separated by a single instance of a global communication involving all the processes performing the calculations. These together with a specialised, but flexible data distribution, are the major hallmarks of the framework. The two stages of the computation can be further parallelised and/or optimised, and we discuss two examples of potential intra-node parallelisation approaches below. We note that the framework as such does not imply what algorithm should be used by the MPI processes, e.g., [16], though of course its overall performance and scalability will depend on the specific choice.

4.1.1. Data distribution and algorithm The simple structure of the top-level parallelism outlined above is facilitated by a data layout which assumes that both the map s and its harmonic coefficients $a_{\ell m}$ are distributed. The map distribution is ring-based, that is we assign to each process a part of the full map consisting of the complete rings of pixels. To do so we make use of the assumed equatorial symmetry of the pixelization schemes and first divide all rings of one of the hemispheres (including the equatorial ring if present) into disjoint subsets of consecutive rings, and then each process is assigned one of the ring subsets together with its symmetric counterpart from the other hemisphere. If we denote the rings mapped on a processor i as \mathcal{R}_i , then $\cup_{i=0}^{n_{procs}} \mathcal{R}_i = \mathcal{R}$, where \mathcal{R} is the set of all the rings and n_{procs} is the number of processes used in the computation. We also have $\mathcal{R}_i \cap \mathcal{R}_j = \emptyset$ if $i \neq j$. An example ring distribution is depicted in Figure 3(a), where different colours mark pixels on iso-latitudinal rings mapped on two different processes.

In order to distribute the harmonic domain object we divide the 2-dimensional array of the coefficients $a_{\ell m}$ with respect to a number, m , and assign to each process a subset of values of m , \mathcal{M}_i , and a corresponding subset of the harmonic coefficients, $\{a_{\ell m} : m \in \mathcal{M}_i \text{ and } \ell \in [m, \ell_{max}]\}$. As with the ring subsets, the subsets of m values have to be disjoint, $\mathcal{M}_i \cap \mathcal{M}_j = \emptyset$ if $i \neq j$, and all together include all the values, $\cup_{i=0}^{n_{procs}} \mathcal{M}_i = \mathcal{M}$. An example of the harmonic domain distribution is depicted in Figure 3(b).

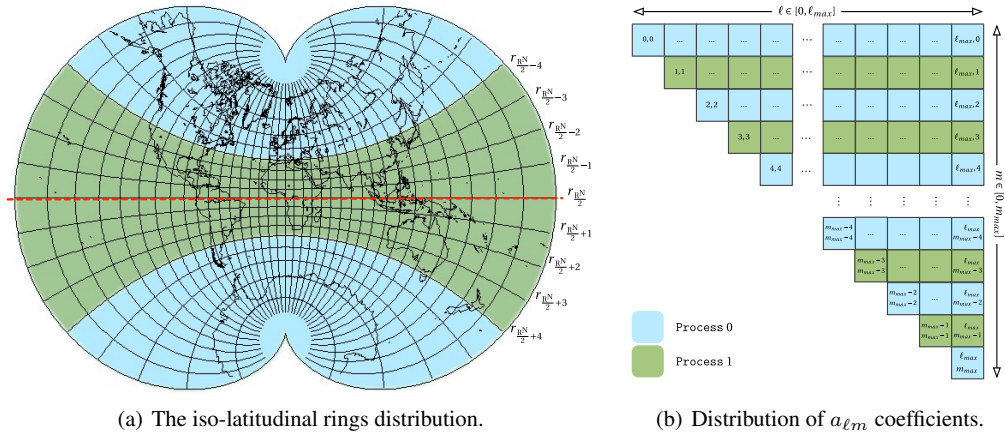


Figure 3. Example of the distribution of the iso-latitudinal rings and the coefficients $a_{\ell m}$ among two processes. Colours mark elements assigned to a single process. For rings visualisation we used an August's Conformal Projection of the sphere on a two-cusped epicyclic.

This data layout allows to avoid any redundancy in terms of computation and memory, and any need for an intermediate inter-process communication. At the same time it is general enough that at least in principle can accommodate an entire slew of possibilities as driven by real applications. Any freedom in selecting a specific layout should be used to ensure a proper load balancing in terms of both memory and work, and it may, and in general will, depend on what specific algorithm is used by each of the MPI processes. Though perfect load balancing, memory- and work-wise, may not be always possible, in practice we have found that good compromises usually exist. In particular, the map distribution which assigns the same number of rings to each process is typically satisfactory, even if it may in principle lead to an unbalanced work and/or memory distribution. The latter is indeed the case for the HEALPIX pixelization, which does not preserve the same number of pixels per ring, what introduces differences in the memory usage between the processes but also in the workload. Some fine tuning could be used here to improve on both these aspects, as it is indeed done in S²HAT, but even without that the differences are not critical. For the distribution of the harmonic objects, each process i stores a set of m values defined as, $\mathcal{M}_i \equiv \{m : m = i + k n_{procs} \text{ or } m = m_{max} - i - k n_{procs}, \text{ where } k \in [0, m_{max}/2]\}$, i.e., values of m equal to $[i, i + n_{procs}, \dots, m_{max} - i - n_{procs}, m_{max} - i]$. This distribution appears satisfactory in a number of cases as it is explained in the next section.

We note that this data layout imposes an upper limit on the number of processes which could be used, given by $\min(m_{max}/2, \mathcal{R}_N/2)$ (we assign at least two rings and two m values to each process). This is typically sufficient, as usually we have $\mathcal{R}_N \propto m_{max} \simeq \ell_{max}$, and the limit on the number of processes is $\mathcal{O}(\ell_{max})$.

With this data distribution, Algorithms 1 and 2 require only straightforward modifications to become parallel codes. As an example, Algorithm 3 details the parallel inverse transform, which is a parallel version of Algorithm 2. The operations listed there are to be performed by each of the n_{proc} MPI processes involved in the computation. Like its serial counterpart, the parallel algorithm can be also split into steps. First, we precompute the starting values of the Legendre recurrence μ_m (9), but only results for $m \in \mathcal{M}_i$ are preserved in memory. In next step, we calculate the functions Δ_m^A using equation (12) for every ring r of the sphere, but only for m values in \mathcal{M}_i . Once this is done, a global (with respect to the number of processes involved in the computation) data exchange is performed so that at the end each process has in its memory the values of $\Delta_m^A(r)$ computed only for rings $r \in \mathcal{R}_i$ and for all m values. Finally, via FFT we synthesise the partial results as in equation (11). From the point of view of the framework, steps 1 and 2 constitute the first stage of the computation, followed by the global communication, and the second computation involving here a calculation of the FFT. As mentioned earlier, though the communication is an inherent part

Algorithm 3 GENERAL PARALLEL alm2map ALGORITHM (CODE EXECUTED BY EACH MPI PROCESS)

Require: $\alpha_{\ell m} \in \mathbb{C}^{\ell \times m}$ for $m \in \mathcal{M}_i$ and all ℓ

Require: indices of rings $r \in \mathcal{R}_i$

STEP 1 - PRE-COMPUTATION

for every $m \in \mathcal{M}_i$ **do**

◦ μ_m pre-computation equation (9)

end for (m)

STEP 2 - Δ_m^A CALCULATION

for every $m \in \mathcal{M}_i$ **do**

◦ precompute recurrence coefficients, $\beta_{\ell m}$ equation (8)

for every ring $r \in \mathcal{R}$ **do**

◦ initialise the recurrence: $\mathcal{P}_{mm}, \mathcal{P}_{m+1,m}$ using precomputed μ_m equations (9) & (10)

for every $\ell = m + 2, \dots, \ell_{max}$ **do**

◦ compute $\mathcal{P}_{\ell m}$ via the 2-point recurrence, given precomputed $\beta_{\ell m}$, equation (7)

◦ update $\Delta_m^A(r)$, equation (12)

end for (ℓ)

end for (r)

end for (m)

GLOBAL COMMUNICATION

◦ $\{\Delta_m^A(r), m \in \mathcal{M}_i, \text{all } r \in \mathcal{R}\} \xrightarrow{\text{MPI_Alltoallv}} \{\Delta_m^A(r), r \in \mathcal{R}_i, \text{all } m \in \mathcal{M}\}$

STEP 3 - s_n CALCULATION

for every ring $r \in \mathcal{R}_i$ **do**

◦ via FFT calculate $s(r, \phi)$ for all samples ϕ of ring r , given pre-computed $\Delta_m^A(r)$ for all m .

end for (r)

return s_n for all ($r \in \mathcal{R}_i$)

of the framework, the two computation stages can be adapted as needed. Hereafter, we adhere to the standard SHT algorithm as used in Algorithm 3.

4.1.2. Performance analysis

Operation count. We note that the scalability of all the major data products can be ensured by adopting an appropriate distribution of the harmonic coefficients and the map. This stems from the fact that apart of the input and output objects, i.e., the maps and the harmonic coefficients themselves, the volumes of which are inversely proportional to the number of the MPI processes by construction. The largest data objects derived as intermediate products are the arrays Δ_A and Δ_S , whose sizes are $\mathcal{O}(\mathcal{R}_N m_{max}/n_{proc})$, and thus again decreasing inversely proportionally with n_{procs} . These objects are computed during the first computation stage, then redistributed during the global communication step, and used as the input for the second stage of the computation. Their total volume, as integrated over the number of processes, therefore also determines the total volume of the communication, as discussed in the next section. As these three objects are indeed the biggest memory consumers in the routines, the overall memory usage scales as $(n_{pix} + 2 m_{max} \ell_{max} + \mathcal{R}_N m_{max})/n_{procs}$. For typical pixelization schemes and full sky maps analysed at its sampling limit, we have $n_{pix} \sim \ell_{max}^2$, $\ell_{max} \simeq m_{max}$, and $\mathcal{R}_N \sim n_{pix}^{1/2}$, and therefore each of these three data objects contribute nearly evenly to the total memory consumption of the routines. Consequently, the actual high memory water mark is typically $\sim 50\%$ higher than the volume of the input and output data. We note that if all three parameters, ℓ_{max} , m_{max} , \mathcal{R}_N , can have arbitrary values, then reusing the memory, and, in particular an *in-place* implementation of the

transforms is not straightforward, as it may not be easy or possible to fit any of these data objects in the space assigned for other data objects. For this reason, such options are not implemented in the software discussed here. Given the assumed data layout, also the number of flops required for the evaluation of FFTs and associated Legendre functions scales inversely proportionally to the number of the processes. The only exception is related to the cost of the initialisation of the recurrence, equation (9), which is performed redundantly by each MPI process to avoid extra communication. As each process receives at least one value of m on order of m_{max} , it has to perform at least $\mathcal{O}(m_{max})$ operations as part of the pre-computation, equation (9). A summary of the flops required on each step of the parallel SHT algorithms is presented in Table I, where we have assumed here that the subsets \mathcal{R}_i and \mathcal{M}_i contain $\simeq \mathcal{R}_N/n_{proc}$ and $\simeq m_{max}/n_{proc}$ elements, respectively.

		<i>Flops</i>
PRE-COMPUTATION	STEP {2,1}	$\mathcal{O}(m_{max})$
RECURRENCE	STEP {3,2}	$\mathcal{O}\left(\mathcal{R}_N \ell_{max} \frac{m_{max}}{n_{proc}}\right)$
FFTs	STEP {1,3}	$\mathcal{O}\left(\frac{\mathcal{R}_N}{n_{proc}} m_{max} \log m_{max}\right)$

Table I. Estimates of floating point operations required in the parallel SHT algorithm.

Communication cost. The overall performance of the algorithms depends also on the communication cost. The data exchange is performed by using a single collective operation of the type all-to-all, in which each process sends a distinct message to every other process. Hereafter, we assume that the time needed to send a message of S_{msg} bytes between any two MPI process can be estimated by $\alpha + \beta S_{msg}$, where α is the latency for sending a message and β is the inverse of the bandwidth. Different MPI libraries use different specific algorithms to implement such a communication, which moreover may be different for different sizes of the message. For concreteness, we base our analysis on MPICH [27], a widely used MPI library. Similar considerations can be performed for other libraries. For short messages (≤ 256 kilobytes) where the latency is an issue, the all-to-all communication in MPICH [28] uses the index algorithm by Bruck et al. [29]. This algorithm features a trade-off between the startup time and the data transfer time, and while taking only $\log n_{proc}$ steps, it communicates data volumes potentially as big as $S_{msg}(n_{proc}/2) \log n_{proc}$. For long messages and an even number of processes, MPICH switches to a pairwise-exchange algorithm, which performs series of exchanges among pairs of processes. In the case of uniform distributions of the rings r and multipoles m between processes, the size of the message to be exchanged between each pair of processes is given as,

$$S_{msg} := \mathcal{R}_N \frac{m_{max}}{n_{proc}} n_{\mathbb{C}}, \quad (16)$$

where $n_{\mathbb{C}}$ is the size in bytes of a complex number representation (usually $n_{\mathbb{C}} = 16$). Using this expression and all the assumptions listed above, the following formula estimates the total time required for communication in our parallel SHT algorithm,

$$T_{comm} = \begin{cases} \alpha \log n_{proc} + \beta S_{msg} \frac{n_{proc}}{2} \log n_{proc}, & \text{when } S_{msg} \leq 256 \text{ kB,} \\ \alpha (n_{proc} - 1) + \beta S_{msg} (n_{proc} - 1), & \text{when } S_{msg} > 256 \text{ kB.} \end{cases} \quad (17)$$

In Figure 4 we illustrate theoretical runtimes depending on the number of the employed processes (leftmost panel) and on the size of the problem (middle panel). The transform parameters have been set assuming a standard full sky, fully resolved case, and the HEALPix pixelization, i.e., $\ell_{max} = m_{max} = 2 N_{side}$, $\mathcal{R}_N = 4 N_{side} - 1$, $n_{pix} = 12 N_{side}^2$. Here, N_{side} is the HEALPix resolution

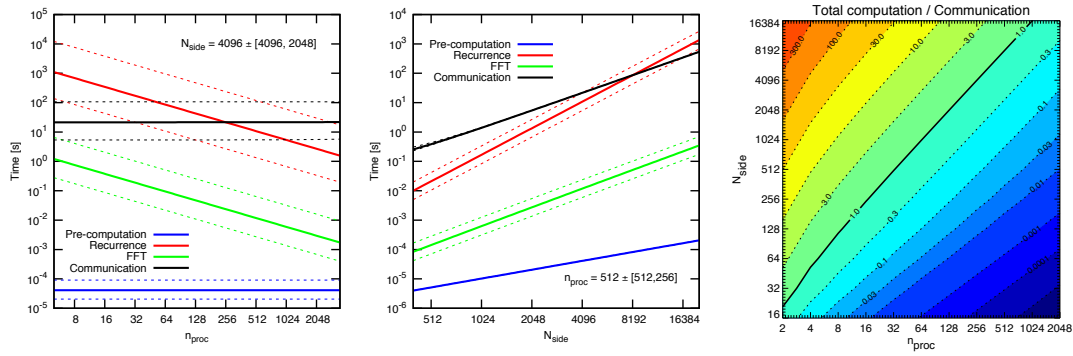


Figure 4. Two plots on the left show theoretical runtimes (logarithmic scale) as function of the number of processes for a fixed size of the problem, leftmost panel, and as a function of the resolution/size of the problem for a fixed number of processes, middle. The sizes of the problem shown in the leftmost panel with thick, solid lines, correspond to $\approx 2 \cdot 10^8$ pixels and a similar number of the harmonic modes, corresponding to the HEALPix parameter value $N_{side} = 4096$. Thin, dashed lines show cases with 4 times fewer (more) pixels, lower and upper lines respectively. In the middle panel the number of processes is assumed to be $n_{proc} = 512$, thick, solid line, or 256 (1024), as shown by thin dashed lines. The contours in the rightmost panel display a ratio of the total computation to communication times shown as a function of the number of process and the size of the problem.

parameter (for more details see beginning of section 5 and [22]). For the hardware parameters, in the estimation of the communication cost we have used, following [30, 31], $\alpha = 10^{-5}$, $\beta = 10^{-9}$, while for the cost of calculations we have assumed that each MPI process attains the effective performance of 10 Gflops per second, resulting in a time of $\gamma = 10^{-10}$ seconds per flop. The choice we have made is meant to be only indicative, yet at its face value the latter number is more typical of the MPI processes being nodes of multi-cores than just single cores, in particular given the usual efficiency of such calculations, which is usually on the order of 10 – 20%. If less efficient MPI processes are used, then the solid curves in Figure 4 corresponding to the computation times should be shifted upwards by a relevant factor.

The dominant calculation is the recurrence needed to compute the associated Legendre functions in agreement with our earlier numerical result, Figure 2. The computation scales nearly perfectly with the number of the used processes, decaying as $\propto 1/n_{proc}$ while increasing with the growing size of the problem, $\propto \ell_{max}^2 \propto N_{side}^2$. At the same time, the communication cost is seen to be independent of the number of processes. This is because for the numbers of processes considered here, the size of the message exchanged between any two processes is always large and the communication time is thus described by the second term of the second equation of the set of equations (17). Given that the single message size, S_{msgs} , decreases linearly with the number of processes, the total communication time does not depend on it. The immediate consequence of these two observations is that both these times will be found comparable if a sufficiently large number of processes is used. A number of processes at which such a transition takes place will depend on the constants entering in the calculation of equations (17) and the size of the problem. This dependence of the critical number of processes on the size of the problem is shown in the rightmost panel of Figure 4, with thick solid contour labelled 1.0. Clearly, from the perspective of the transform runtimes, there is no extra gain which could be obtained from using more processes than given by their critical value. In practice, the number of processes may need to be fixed by memory rather than by computational efficiency considerations. As the memory consumption depends on the problem size, i.e., n_{pix} , in the same way as the communication volume, by increasing the number of processes in unison with the size of the problem we can ensure that the transform runs efficiently, i.e., the fraction of the total execution time spent on communicating will not change, and that it will always fit within the machines memory.

We note that some improvement in the overall cost of the communication could be expected by employing a non-blocking communication pattern applied to smaller data objects, for example

single rows of the arrays Δ_A and Δ_S corresponding to a single ring or m value, successively and immediately after they are computed. Such an approach could extend the parameter space in which the computation time is dominant and the overall scaling favorable, by overlapping communication with some computation. We expect this to be beneficial only for the inverse transform as for the direct one the computation preceding the communication, and involving the FFTs, is subdominant with respect to both the communication and the recurrence time in the regime where these both are becoming comparable. Therefore, the gain from overlapping the FFT calculation with the communication could be at the best minor. Moreover, even for the inverse transform the gain would be rather limited as long as the same, large, total volume of data, is required to be communicated and the communication time will eventually supersede that of the calculation at some concurrency. We note that the communication volume could be decreased, and thus again the parameter space with the computation dominance extended, if redundant data distributions and redundant calculations are allowed for in the spirit of the currently popular *communication avoiding* algorithms, e.g., [32]. Though these kinds of approaches are clearly of interest they are outside the scope of this work and are left here for future research.

We also note that the conclusions may disfavour algorithms which require big memory overhead and/or need to communicate more. Such algorithms will tend to require a larger number of processes to be used for a given size of the problem, and thus will quickly get into the bad scaling regime. Their application domain may be therefore rather limited. We emphasise that this general conclusion should be reevaluated case-by-case for any specific algorithm. Likewise, it is clear that the regime of the scalability of the standard SHT algorithm could be extended by decreasing the communication volume. Given that the communicated objects are dense, the latter would typically require some lossy compression techniques and therefore we do not consider them here. Instead we focus our attention on accelerating intra-node calculations. This could improve the transforms runtimes for a small number of processes. However if successful, unavoidably it will also result in lowering the critical values for the processes numbers, leading to losing the scalability earlier. Nevertheless, the overall performance of these improved transforms would never be inferior to that of the standard ones whatever the number of processes.

4.2. Exploiting intra-node parallelism

In this section we consider two extensions of the basic parallel algorithm, Algorithm 3, each going beyond its MPI-only paradigm. Our focus hereafter is therefore on the computational tasks as performed by each of the MPI processes within the top-level framework outlined earlier. Guided by the theoretical models from the previous section, we expect that any gain in performance of a single MPI process will translate directly into analogous gain in the overall performance of the transforms, at least as long as the communication time is subdominant, that is in a relatively broad regime of cases of practical importance. Hereafter we therefore develop implementations suitable for multi-core processors and GPUs and discuss the benefits and trade-offs they imply.

As highlighted in Figure 4, the recurrence used to compute the associated Legendre functions takes by far a dominant fraction of the overall runtime of the three computational steps. Consequently, we will pay special attention to accelerating this part of the algorithm, while we keep on using standard libraries to compute the FFTs. The latter will be shown to deliver sufficient performance given our goals here. The associated Legendre function recurrence involves three nested loops and we will particularly consider advantages and disadvantages of their different ordering from the perspective of attaining the best possible performance. We note the recurrence is involved on the stages when either the harmonic coefficients $a_{\ell m}$ are computed given the already available Δ_m^A in the case of the direct transform, or when the computation of Δ_m^S is performed given the input coefficients $a_{\ell m}$. In both cases, each shared-memory node stores information concerning all observed rings and a subset of m values as defined by \mathcal{M}_i and as distributed on the memory-distributed level of parallelisation. Hereafter, for simplicity we will refer to the algorithms for the direct and inverse spherical transforms by the core names of their corresponding routines, `map2alm` and `alm2map` respectively.

4.2.1. Multithreaded version for multi-core processors. Since the largest and the fastest computers in the world today employ both shared and distributed memory architectures, their potential can not be fully exploited by software which uses only distributed memory approaches such as MPI. To overcome this problem, hybrid approaches, mixing MPI and shared memory techniques such as multithreading have been proposed. In this spirit, we introduce a second level of parallelism based on the multithreaded approach in the computation of SHT.

We note that the three nested loops which we need to perform to calculate the transforms are in principle interchangeable. In practice, the loop over ℓ has to be the innermost one in order to avoid significant memory overhead or repetitive re-calculations of the same objects. Out of the two remaining loops, selecting the loop over m as the outermost, as in the MPI-only case from Algorithm 3, allows precomputing $\beta_{\ell m}$, equation (8), \mathcal{P}_{mm} , equation (9), and $\mathcal{P}_{m+1,m}$, equation (10), only once for each ring. This also requires extra storage to store $\beta_{\ell m}$ for a given m , which is however only on the order of $\mathcal{O}(\ell_{max})$, as well as the intermediate results, i.e., $\Delta_m^{A/S}$, which require additional storage on the order of $\mathcal{O}(\ell_{max} \mathcal{R}_N)$. This latter estimate is comparable to the size of the input and output, and therefore not negligible in the memory budget of the calculation. Nevertheless, given that it decreases with the number of MPI process and given the sizes of memory banks of modern supercomputers, tens of Gigabytes per shared memory node, this can be managed in practice straightforwardly, as has been indeed demonstrated earlier in the case of the MPI-only routines. This choice of the loop ordering is therefore efficient from both computation and memory points of view. Consequently, in the multithreaded versions of Algorithms 1 and 2 the outermost loop, parallelised using OpenMP, is the loop over m . The set of m values, assigned to each MPI process i , i.e., \mathcal{M}_i , is then evenly divided into a number of subsets equal to the number of threads mapped onto each physical core available on a given multi-core processor. We denote this subset of m values as $\mathcal{M}_i^{T_j}$, where subscript j corresponds to the thread number. In such a way, each physical core executes one thread concurrently and calculates intermediate results of SHT for its local (in respect to the shared memory) subset of m . Algorithm 4 describes STEP 1 of Algorithm 3 in its multithreaded version.

Algorithm 4 Δ_m^A CALCULATION PER THREAD

Require: all $m \in \mathcal{M}_i^{T_j} \Rightarrow \bigcup_{j=1}^{N_{th}} \mathcal{M}_i^{T_j} = \mathcal{M}_i$

for all $m \in \mathcal{M}_i^{T_j}$ **do**

- initialise \mathcal{P}_{mm} and $\mathcal{P}_{m,m+1}$, equations (9)
- precompute $\beta_{\ell m}$, equation (8)
- for** every ring r **do**
- for** every $\ell = m + 2, \dots, \ell_{max}$ **do**
- compute $\mathcal{P}_{\ell m}$ via $\beta_{\ell m}$, equation (7)
- update $\Delta_m^A(r)$ in the shared memory, equation (12)
- end for** (ℓ)
- end for**(r)
- end for**(m)

Load balance between active threads. As already noted for the MPI-only cases, the loop ordering adopted here does not ensure proper load balance between different threads. Nevertheless, it is flexible enough to leave room allowing for appropriate tuning. As before, potential imbalance of the computational load is due to the different number of steps which need to be performed during the recursive calculation of the associated Legendre functions, and which is equal to $\ell_{max} - m + 1$. Therefore it explicitly depends on m . To avoid this problem, a care needs to be taken while defining the specific subsets of m values, which are to be assigned to each thread, $\mathcal{M}_i^{T_j}$. Hereafter, we do so by replicating the same approach as used earlier to balance the workload on the distributed memory level, and generate the subsets, $\mathcal{M}_i^{T_j}$, by taking values from the set \mathcal{M}_i in the "max-min" fashion

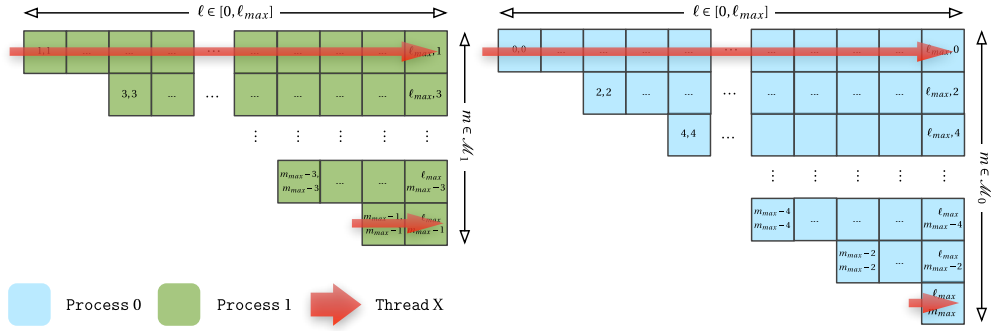


Figure 5. Example of two "min-max" pairs of m values assigned to Thread X on Process 1 (left) and 0 (right) respectively. The read arrow represents the direction and the length of the ℓ -recurrences (equation (10)) evaluated by a thread for a given "min-max" pair.

i.e., sequentially we pick from \mathcal{M}_i the maximal and minimal values as available at any given time and assign them to a chosen thread, while at the same time we try to keep the number of such pairs per thread nearly the same for every thread. Example of such a mapping is depicted in Figure 5.

We note that if such a distribution of m modes is done consistently on both distributed- and shared- memory levels, it will not only result in balancing the workload within the single node but also across all the nodes involved on the MPI level, ensuring global load balance. That such balance indeed follows can be seen immediately by noticing that on the distributed memory level each set \mathcal{M}_i assigned to one of the nodes consists of pairs of m values, $\{m', m''\}$, picked as a min-max pair and the sum of which is constant and given by $m' + m'' = m_{max}$. Consequently, the same numerical cost is incurred in the computation of the associated Legendre functions for each pair, as the latter is completely determined by the number of the recursion steps, which is given by $\ell_{max} - m' + 1 + \ell_{max} - m'' + 1 = 2\ell_{max} - m_{max} + 2$ and therefore independent of both m' and m'' . At the same min-max principle applied on the intra-node level will assign each such pairs to one of the threads and therefore lead to a near perfect load balancing as long as the number of pairs assigned per thread is nearly the same for all the threads. Of course, departures from the well-balanced load may be expected whenever the number of m values assigned per node or per thread becomes small.

4.2.2. SHT on GPGPUs. A CUDA (Compute Unified Device Architecture) device is a chip consisting of a number of Streaming Multiprocessors (SM), each containing hundreds of simple cores (typically 512), called Streaming Processors (SP), which execute instructions sequentially. The Streaming Multiprocessors perform instructions in a SIMD (Single Instruction Multiple Data) fashion and can maintain hundreds of active threads, referred to as a thread block hereafter, which are launched in batches of typically 32 threads, called warps. The warps are executed sequentially in the cyclic order, and this helps to hide the memory latency between the local memory of SMs and the global memory of the GPU. The former memory is shared within the SMs, it is fast but limited. For example it is up to 48KB in size for the latest Nvidia cards. There is no synchronisation between different SMs and they can communicate only through relatively slow global memory.

The fact that all the active threads of the GPU execute the same instruction set suggests that the most suitable approach for this device is to choose the loop over the rings, r , as the outermost one, and to parallelise the calculation with respect to it. This is contrary to the multi-node implementation, as described earlier, which is based on setting the loop over m as the outermost one. The latter choice has the advantage of optimising the number of numerical operations, such as those involved in the calculation of the $\beta_{\ell m}$ object, which is independent on r and can be computed for each value of m as set by the outermost loop and used for all values of r in the loop over r . The downside of this approach is that if the outermost loop is parallelised and different values of m assigned to different threads, then the innermost loop over ℓ number will require a different number

of passes depending on the specific value of m . Consequently, the parallelisation on the finest level, i.e., within each warp, will be lost. Setting the loop over r as the outermost not only avoids this problem but also permits using more active threads, as at this stage of the calculation each GPU will store in its memory the object $\Delta_m(r)$ distributed between the GPUs with respect to m , with each GPU storing only those of its entries which correspond to the subset \mathcal{M}_i of all values of m but for all values of r ($\in \mathcal{R}$). These advantages seem to offset the increased computational load as involved in the re-calculation of $\beta_{\ell m}$, equation (9), and we use therefore this approach to implement the SHT algorithms on GPUs. The respective algorithm then involves, first, dividing the set of all rings into disjoint subsets, and subsequently assigning them to different threads of every GPU used in the computation. The threads perform then the calculation for all the rings assigned to it as well as all values of m from the subset, \mathcal{M}_i , as assigned to this GPU (and therefore common to all its threads), Figure 6. The overall load balancing of the calculation is ensured if the number of rings per thread is the same for all threads, and the values of m are distributed between the GPUs in the same way as in the case of the multi-node algorithm. We note that a similar approach has been adopted in [21] in the case of their `alm2map` routine, and we will hereafter use their solution to optimise the calculation of $\beta_{\ell m}$. This approach achieves the best performance if one ring is assigned per GPU thread, as then the involved recalculation can be completely avoided.

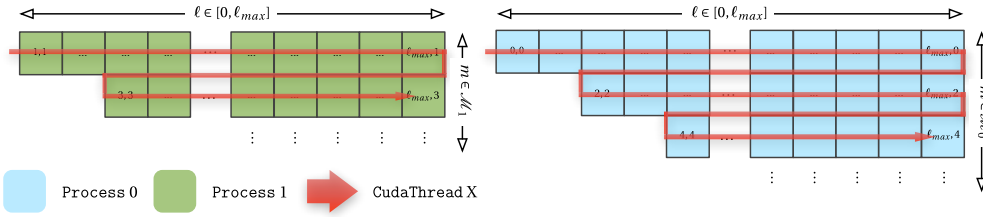


Figure 6. Schematic view of the iterations of the recurrence with respect to ℓ (equation 10) evaluated by each thread on CUDA device assigned to two different GPUs, referred to as Process 1 and 0 respectively.

Setting the loop over the rings as the outermost one enables two completely different approaches for the memory management in the direct and inverse SHT. We describe them in the following two paragraphs.

map2alm. Mapping the rings onto a big number of threads facilitates the parallelisation of the SHT algorithm on GPUs, as it permits each active thread to perform exactly the same number and type of operations but with different values. However, in the case of the direct transform each thread calculates a contribution to all non-zero values of the harmonic coefficient matrix $a_{\ell, m}$ for $m \in \mathcal{M}_i$ and $\ell \in [0, \dots, \ell_{max}]$, and coming from a subset of all rings as assigned to this thread. The calculation of the final result therefore requires an all-reduce operation involving all the active threads within the GPU. As on CUDA the active threads can communicate only within the thread block, the task is not completely straightforward.

Algorithm 5 outlines STEP 3 of Algorithm 1 adapted to CUDA. The set of values of m to be processed by a selected GPU is given by the set \mathcal{M}_i , assigned to each GPU on the distributed-memory level. To take advantage of the fast shared memory, and to simultaneously avoid the high latency memory of the device, we first calculate the values of $\beta_{\ell m}$ vectors in segments, which can fit in the shared memory i.e., all threads within a block in a "collaborative" way compute in advance $\beta_{\ell m}$ values and store them in shared memory. Next, these precomputed values are used to evaluate an associated Legendre function $\mathcal{P}_{\ell m}$, followed by each active thread fetching from global memory the precomputed sum $\Delta_m^S(r)$ and performing the final multiplication from equation (13). Once it is done, we sum all these partial results within a given block of threads using an adapted** version

**We omitted a part of the code, which writes the partial result to global memory

of the parallel reduction primitive from CUDA SDK. We note that for typical values of ℓ_{max} , the memory needed to store all the harmonic coefficients $\mathbf{a}_{\ell m}$ exceeds the shared memory available in each SM. They will have to be therefore communicated to the global memory in batches with sizes close to that of the shared memory. Consequently at any time all the thread blocks will send the data concerning the same subset of the harmonic coefficients and do so multiple times. This implies that as many as NBT threads, where NBT is the number of blocks of threads, will try to update the values of the same variables stored in the same space of the global memory. This leads to a *race condition problem*. The race conditions can be resolved in CUDA by using atomic operations (e.g., via CUDA `atomicAdd` function), which are capable of reading, modifying, and writing a value back to the shared or global memory without an interference from any other thread, which guarantees that the race condition will not occur. However, if two threads perform an atomic operation at the same memory address at the same time (which may be the case in our algorithm), those operations will be serialized i.e., in this situation each atomic operation will be done one at a time, but execution will not progress past the atomic operation until all the threads have completed it. Consequently, atomic operations may cause dramatic degrade of the performance which additionally strongly depends on the hardware^{††}. Alternately, we can avoid the race conditions and thus, execution of the expensive atomic functions by increasing granularity of the recurrence step. In this scenario we loop over m values on CPU and execute as many *small* CUDA kernels as elements in \mathcal{M}_i , which for given values m and μ_m , evaluate NBT reduced (per block of threads) results $a_{\ell m}^*$, which are next copied from the device to the host memory and summed together on CPU. This variant of our algorithm corresponds to the state `HYBRID_EVALUATION = true`, in Algorithm 5. This clearly could be the only option if the atomic operations are not supported on the GPUs used for the calculation. We note however that even if they are supported, this latter approach may prove to be more efficient, as it indeed was the case for the GPU used in this work. For this reason, all the experimental results discussed in the next section are calculated with the reduction performed on a CPU.

`alm2map`. The inverse SHT also requires a reduction of the partial results, but contrary to the direct transform, the summation is done over ℓ as shown by equation (12). With the rings assigned to single threads, this leads to a rather straightforward implementation. Algorithm 6 outlines the `alm2map` algorithm devised for GPUs and it is based on the version detailed in [21].

First, we partition the vector of the coefficients $\mathbf{a}_{\ell m}$ in global memory into small tiles. In this way a block of threads can fetch resulting segments in a sequence and fit them in shared memory. Next, we calculate $\beta_{\ell m}$ values in the same way as in the `map2alm` algorithm. With all the necessary data in fast shared memory, in the following steps, each thread evaluates in parallel an associated Legendre function and updates the partial result Δ_m^A .

There are two major differences with the `map2alm` algorithm discussed earlier. First, the updates are performed without the race condition problem as each thread computes, stores, and transfers to global memory values which are specific to it and independent on the values calculated by the other threads. Second, there is only one value per thread, and therefore at most NTB values per thread block, which need to be sent from any SM to global memory, what is typically much smaller than the $\mathcal{O}(\ell_{max})$ volume required in the case of `map2alm`. We will return to these conclusions while discussing our numerical experiments later on in this paper.

4.2.3. Specific optimisations for GPU. In addition to the optimisations specific to our application and described in the previous section, the algorithms developed here were optimised following the general programming rules and guidelines as developed for CUDA GPGPUs, e.g., [34, 35]. For completeness we briefly list the most consequential of them here. We have limited the communication between the host and the device to the minimum, with the exception of our alternative solution to the race condition problem for `map2alm`. We have expressly avoided asymmetrical branching in the control flow on the GPU itself. We have minimised the need to access

^{††}For instance, Nvidia reports [33] that there is a substantial difference between atomic operation performance between CUDA with compute capability 1.1 (where it first appeared) and more recent Fermi devices (compute capability 2.x).

Algorithm 5 $a_{\ell m}$ CALCULATION ON CPU&GPU

Require: Δ_m^S & μ_m vectors in the GPU global memory for $m \in \mathcal{M}_i$
for every ring r assign to one GPU thread **do**
 for every $m \in \mathcal{M}_i$ **do**
 for every $\ell = m + 2, \dots, \ell_{max}$ **do**
 • use precomputed or, if needed, precompute in parallel a segment of $\beta_{\ell m}$, equation (8);
 • compute $\mathcal{P}_{\ell m}$ via equation (7);
 • evaluate: $a_{\ell m}^r = \Delta_m^S(r) \mathcal{P}_{\ell m}(r)$
 • sum all partial results $a_{\ell m}^r$ within a block of threads

$$a_{\ell m}^* = \sum_{i=0}^{NTB} (a_{\ell m}^r)_i \quad (18)$$

and store the result in the shared memory.
if the shared memory is filled **then**
 if HYBRID_EVALUATION **then**
 • for each block of threads with different id = BID, write $a_{\ell m}^*$ in the device global memory

$$a_{\ell m}^{GPU}[\text{BID}] = a_{\ell m}^*$$

• copy partial results $a_{\ell m}^{GPU}[\dots]$ from the device to the host memory and sum together using CPU
 for $i = 0 \rightarrow \text{NBT}$ **do**
 $a_{\ell m} = a_{\ell m} + a_{\ell m}^{GPU}[i]$
 end for
 else
 • update $a_{\ell m}$ in the global memory via an atomic addition: $a_{\ell m} = a_{\ell m} + a_{\ell m}^*$
 end if
 end if
 end for (ℓ)
 end for (m)
end for (r)
return $a_{\ell, m}$: $m \in \mathcal{M}_i$ and $\ell \in [2, \dots, \ell_{max}]$

Algorithm 6 Δ_m^A CALCULATION ON GPU

Require: $a_{\ell, m} \in \mathbb{C}^{m_{max} \times \ell_{max}}$ matrix in GPU global memory
for every ring r assign to one GPU thread **do**
 for $m \in \mathcal{M}_i$ **do**
 for every $\ell = m + 2, \dots, \ell_{max}$ **do**
 • use fetched or, if needed, fetch in parallel a segment of $a_{\ell m}$ data;
 • use precomputed or, if needed, precompute in parallel a segment of $\beta_{\ell m}$, equation (8);
 • compute $\mathcal{P}_{\ell m}$ via equation (7);
 • update $\Delta_m^A(r)$ for a given prefetched $a_{\ell m}$ and computed $\mathcal{P}_{\ell m}$, equation (12);
 end for (ℓ)
 • save final $\Delta_m^A(r)$ in global memory
 end for (m)
end for (r)
return vector of partial results: $\Delta_m^A \in \mathbb{C}$

the global memory by introducing buffering and recalculating or reusing data, e.g., see Algorithm 6. We have also removed, whenever possible, branches in performance-critical sections to avoid warp serialisation.

We note that this kind of optimisations have been studied in the case of the inverse SHT performed on a single GPU by [21]. Our experiments essentially confirm their findings, which we found applicable to both direct and inverse transforms. We therefore refer the reader to their work for a detailed discussion.

4.3. Exploitation of hybrid architectures

Many new supercomputers are heterogeneous, combining computational nodes interconnected with multiple GPUs. This way, one or more multi-core processors have access to one GPU. In such a case, the front end user of the spherical harmonic package will have to decide which type of architecture to choose for the calculation. The three-stage structure of our algorithms allows mapping different steps of the algorithms on either multi-core processors or accelerators. The intention of this flexibility is to be able to assign each step of the algorithm to the architecture on which its performance is better. For example, Nvidia provides its own library for computing FFTs on GPUs i.e., CUFFT [36], whose efficiency strongly depends on the GPU device and on the version of the routine. However, there are cases when highly tuned FFTs perform better if they are executed on multi-core processors. Note that our algorithms require the evaluation of FFTs on vectors of complex numbers, whose length may vary from ring to ring. The default assignment for both transforms is depicted in Figure 7. Due to their sequential nature, we omit steps in which we evaluate the starting values of the recurrence described in equation (9) (STEP 2 in Algorithm 1 and STEP 1 in Algorithm 2), which are always computed on the CPU.

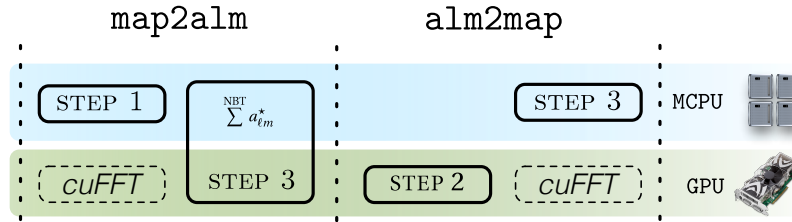


Figure 7. Default assignment of the major steps of the SHT algorithm to one of the two architectures of a heterogeneous system composed of multi-core processors and GPUs. STEP 3 in *map2alm* algorithm may involve any of the two architectures due to the optional use of CPU for reducing partial results between the execution of CUDA kernels.

5. NUMERICAL EXPERIMENTS

In order to validate our new algorithms as well as to evaluate their performance in terms of their accuracy, scalability, and efficiency, we conduct a number of experiments involving a comprehensive set of test cases with a different, assumed geometry of the 2-sphere grid and different values of ℓ_{max} and m_{max} . Each experiment involves first an inverse and then a direct SHT, and uses as input a set of uniformly distributed, random coefficients, $\mathbf{a}_{\ell m}^{init}$, with values in the range $(-1, 1)$. To measure the accuracy of the algorithm we calculate the error as a difference between the final output $\mathbf{a}_{\ell m}^{out}$ and the initial set $\mathbf{a}_{\ell m}^{init}$ via the following formula,

$$\mathcal{D}_{err} = \sqrt{\frac{\sum_{\ell} \sum_m |\mathbf{a}_{\ell m}^{init} - \mathbf{a}_{\ell m}^{out}|^2}{\sum_{\ell} \sum_m |\mathbf{a}_{\ell m}^{init}|^2}}. \quad (19)$$

In these experiments we use HEALPix grids with a different resolution parameter N_{side} , defining the number of the grid points/pixels, $n_{pix} = 12N_{side}^2$, the number of the rings $\mathcal{R}_N = 4N_{side} - 1$, and the maximum number of points along the equatorial ring, $4N_{side}$. Finally, we also set $\ell_{max} = m_{max} = 2N_{side}$, except for some specific experiments. In the following we compare our algorithms with the transforms as implemented in a publicly available library called **libpsht** [9]. To the best of our knowledge it is the fastest implementation of SHT at this time suitable for astrophysical applications, and it is essentially a realisation of the same algorithm as the one used in our parallel algorithms. The **libpsht** transforms are highly optimised using explicit multithreading based on OpenMP directives, and SIMD extensions (SSE and SSE2), but its functionality is limited to shared memory systems. In all our tests we set the number of OpenMP threads, for both **libpsht** and our multi-threaded code, to be equal to the number of physical cores, i.e., `OMP_NUM_THREADS` = 4.

5.1. Scaling with test case sizes

In this set of experiments we generate a set of initial coefficients, $\mathbf{a}_{\ell m}^{init}$, for different values of ℓ_{max} and N_{side} , and use them as an input for our two implementations of the transforms (MPI/OpenMP and MPI/CUDA) as well as for those included in the **libpsht** library. Our experiments are performed on three different computational systems listed in Table II and are run either on one multi-core processor or on a CPU/GPU node.

<i>CPU</i>	<i>clock speed</i>	<i>GPU</i>	<i>Compilers</i>
CORE I7-960K	3.20 GHz	GEFORCE GTX 480	gcc 4.4.3 / nvcc 4.0
CORE I7-2600K	3.40 GHz	GEFORCE GTX 460	gcc 4.4.5 / nvcc 4.0
XEON X5570	2.93 GHz	TESLA S1070	icc 12.1.0 / nvcc 4.0

Table II. Different Intel multi-core processors and NVIDIA GPUs used in our experiments.

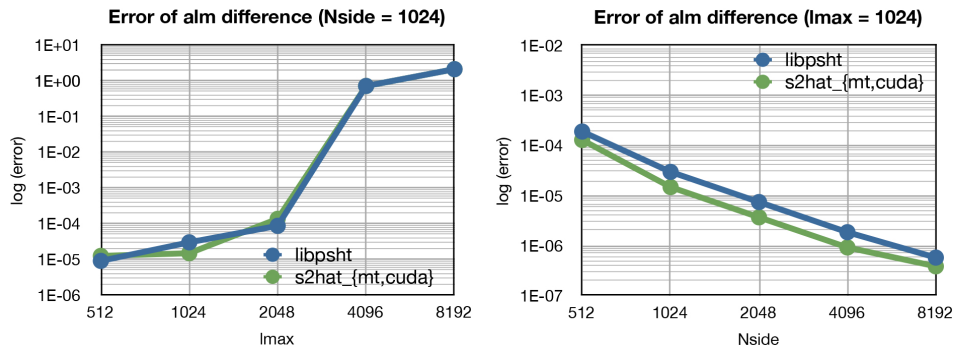


Figure 8. Relative error, \mathcal{D}_{err} , for **s2hat_{mt,cuda}** and **libpsht** transform pairs on the HEALPix grids for different values of ℓ_{max} and N_{side} . Data points show the error values found in our tests. The loss of precision seen in the left panel for $\ell_{max} \gtrsim 2048$ and shared by both implementations is due to aliasing of the high- ℓ power contained in the spatial modes, which are not longer supported by the grid.

Figure 8 displays the error \mathcal{D}_{err} for **s2hat_{mt,cuda}** and **libpsht** transform pairs computed for a set of ℓ_{max} and N_{side} values on the HEALPix grid. Every data point corresponds to the error value encountered in the pairs of transforms. It can be seen that all the implementations provide almost identical precision of the solution in all the cases considered here.

Figures 9 and 10 display the performance of the recurrence part in our algorithms (STEP 3 in Algorithm 1 and STEP 2 in Algorithm 2) in terms of measured time in seconds and number of GFlops per second. The number of flops includes all the operations necessary for maintaining the numerical stability of the recurrences, e.g., the rescaling in the case of S^2HAT routines as described in section 2.

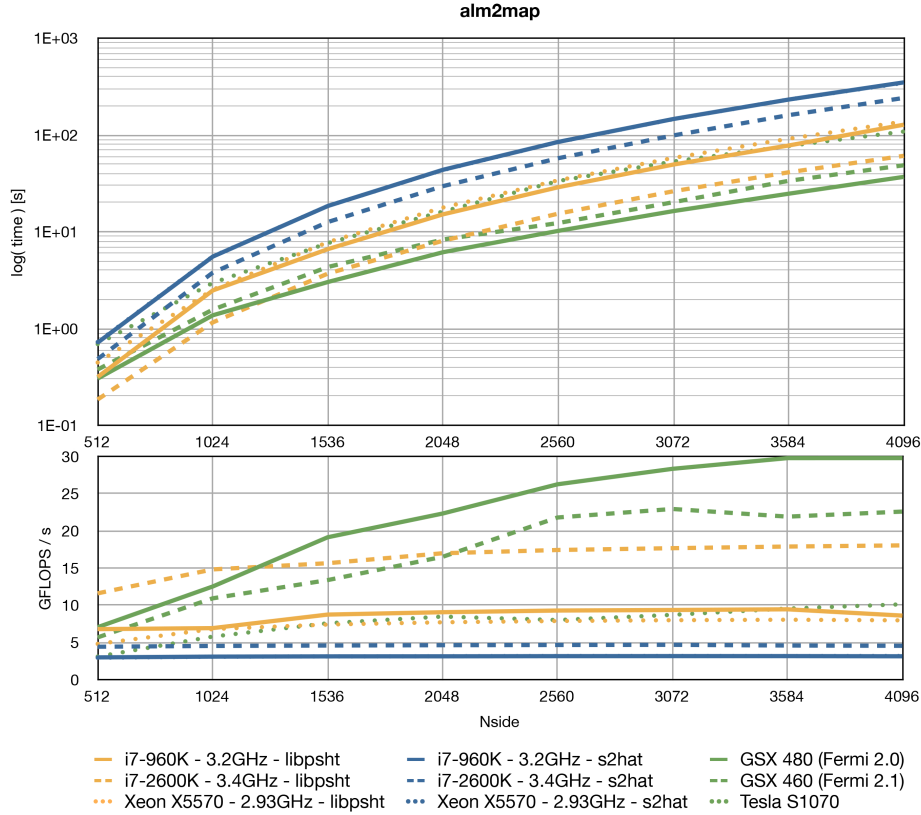


Figure 9. Benchmarks for the different implementations of the inverse SHT performed on either Intel multi-core processors or GPUs, see Table II for details. Results of the **libpsht** code are compared with those of the hybrid S^2 HAT codes (orange, blue and green lines, respectively) and are obtained for the HEALPix grid with $N_{side} = \ell_{max}/2$. The top plot shows timings for calculating Δ_m^A , equation (12), involving the recurrence to calculate the associated Legendre functions. In the case of the CUDA routines, the numbers include time of data movements between CPU and GPU. The bottom plot shows the number of GFlops per second i.e., the total number of flops required in the algorithms divided by the CPU time (see table I). Note that the CUDA routines loose performance in comparison to the SSE optimised **libpsht** due to very costly reduction of partial results, section 4.2.2. Nevertheless, Nvidia GTX 480, green solid line, outperforms **libpsht** on the multicore processor Core i7-960K, solid yellow line. The dotted blue line overlaps perfectly with the solid blue line in both plots.

The resolution of the problem progressively increases from left to right, as do the data objects, memory load, and communication volume, section 4.1.2. The maximal value of $N_{side} = 4096$ was set due to the limitation of the memory of the single GPU used in these tests. We note that for the CUDA routines, the measurements include the time spent in data movements between GPU and CPU.

The expected number of flops in these tests is proportional to N_{side}^3 , and such a scaling is indeed closely followed by the experimental results obtained for the multi-threaded S^2 HAT transforms, which are depicted by the blue lines in the top plot of Figures 9 and 10. The other curves in these plots are however somewhat flatter. This is because these routines become more efficient for the larger data sets. This fact is further emphasised in the bottom plots of these two figures, where the performance of the transform routines is shown in terms of GFlops per second. The lowest, blue curves, corresponding to multithreaded S^2 HAT transforms remain nearly constant across the entire tested range of N_{side} , while the other implementations, and in particular the S^2 HAT-GPU ones, tend to improve their overall efficiency with the size of the test.

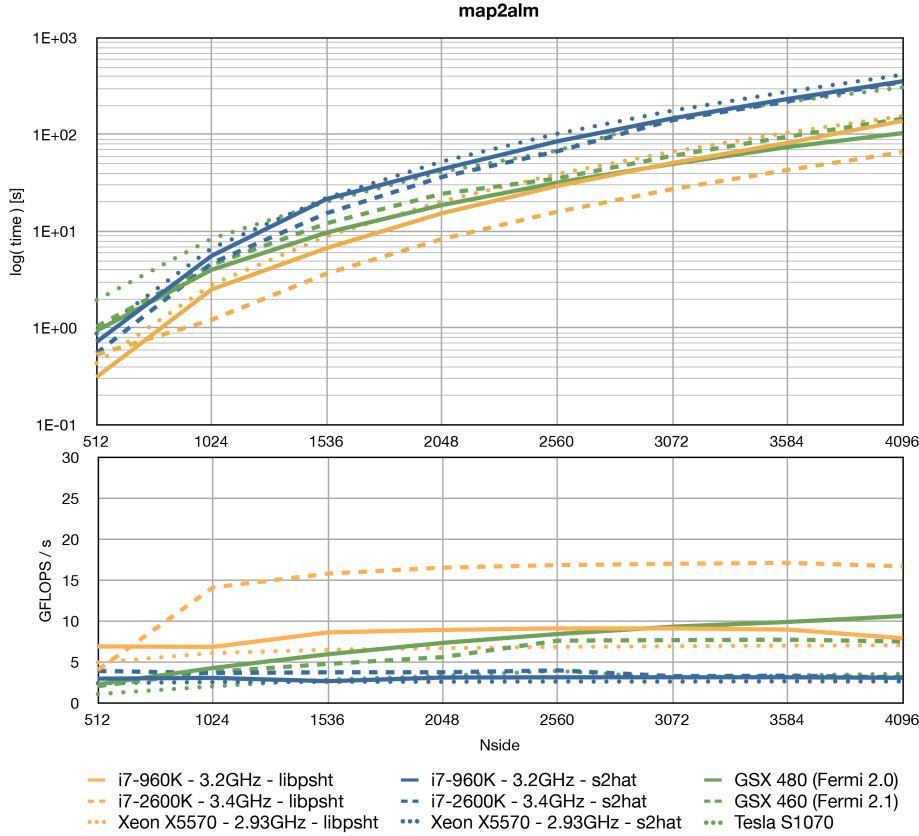


Figure 10. Performance of the implementations of the direct SHT on the systems described in Table II. The top plot shows the time needed for the calculation of Δ_m^S including the \mathcal{P}_{ℓ_m} -recurrence, equation (14). The bottom plot displays the achieved numbers of GFlops per second. For all the runs we use the HEALPix grid with $N_{side} = \ell_{max}/2$. Orange corresponds to the **libpsht** routine, while blue and green to the hybrid S^2HAT ones, multi-threaded and CUDA, respectively.

The fact that multithreaded S^2HAT transforms are systematically slower than **libpsht** by a constant factor could have been expected due to the lack of SSE/AVX intrinsics in our code. These allow performing two or more arithmetic operations of the same kind with a single machine instruction on some hardware devices. Moreover, on the latest intel i7-2600K processor, SSE-optimised **libpsht** implementation further benefits from the new Intel AVX 128-bit instructions set, which have lower instruction latencies, than the standard SSE extension. We note that additional possible optimisation of both libraries could take advantage of extended width of the SIMD register and Advanced Vector Extensions (AVX), which enable operating on 8 float elements per iteration instead of a single one. For the GPU transforms, the following observations can be made:

- The direct transform on CUDA is slower, not only with respect to the corresponding CPU version, but also in comparison with the inverse transform on the same architecture. This is in spite of the fact that on the algebraic level, the calculations, and their implementations, of the objects $\Delta_m^{A/S}$ are essentially the same for both directions of the transform. The performance loss is due to the reduction of partial results, and is incurred first in the shared memory, then in either the slow global memory or the host memory of the CPU, as already discussed on the theoretical level in section 4.2.2.
- Our code on the two Fermi GPUs (GTX 460 & GTX 480) outperforms **libpsht** run on the latest Intel i7 processor in the evaluation of the inverse transform.

- The CUDA routines seem to favour cases with large N_{side} which allow the usage of many threads (as we explained in section 4.2.2, each ring of the sphere is assigned to one CUDA thread) i.e., both GPU routines gain performance with growing number of sky-pixels (thus, also the rings).

Recapitulating, for the inverse transforms we find that the CUDA routine performs better than some of the popular routines for multi-core processors, even if SSE optimisations are used to enhance the performance in these latter cases. For CPUs, the direct and inverse transforms are shown to display similar performance. This is however not the case for the GPUs, where the direct transform is found systematically slower than the inverse one. Nonetheless, the direct transform on GPUs still runs comparably fast to that run on at least some of the current multi-core processors. In both these cases the performance of the CUDA transforms with respect to the multi-core ones, keeps on improving with the sizes of the studied problems.

5.2. Scaling with number of multi-core processors and GPUs

In this section we discuss the performance of our algorithms on clusters of hybrid computers. As before, we set $\ell_{max} = 2 N_{side}$ in all the tests. The results are obtained on CCRT's supercomputer Titane, one of the first hybrid clusters built, based on Intel and Nvidia processors. It has 1,068 nodes dedicated to computation, each node is composed of 2 Intel Nehalem quad-core processors (2.93 GHz) and 24 GB of memory. A part of the system has a multi-GPU support with 48 Nvidia Tesla S1070 servers, 4 Tesla cards with 4 GB of memory and 960 processing units are available for each server. Two computation nodes are connected to one Tesla server by a PCI-Express bus, therefore 4 processors handle 4 Tesla cards. This system provides a peak performance of 100 Tflops for the Intel part and 200 Tflops for the GPU part. The Intel compiler version 12.0.3 and the Nvidia CUDA compiler 4.0 are used for compilation. Furthermore, we define the MPI processes in our experiment such that each of them corresponds to a pair of one multi-core processor and one GPU (4 MPI processes per one Tesla server). To increase intra-node parallelism, we set-up for all runs the number of OpenMP threads equal to the number of physical cores of the Intel Nehalem processor (OMP_NUM_THREADS = 4).

Figure 11 shows the performance on 128 hybrid processors of the computation of a pair of the SHT as a function of the problem size. It can be seen that our both implementations scale well with respect to the problem size, and in all the cases follow closely the predicted scaling $\propto N_{side}$. As already noted in the previous section, only for the inverse transforms we see a performance gain when using GPUs.

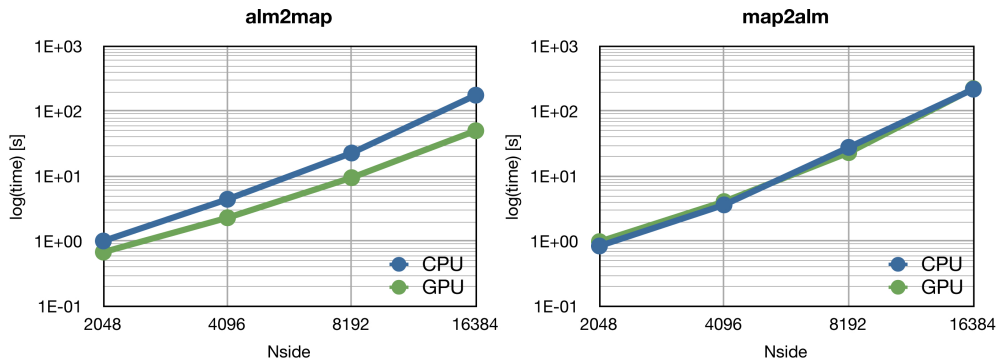


Figure 11. Runtime of the inverse (left), and direct (right), transforms as a function of N_{side} , ($\ell_{max} = 2 N_{side}$). The experiments use 128 MPI processes corresponding to 128 pairs of Intel Nehalem quad-core processors and Nvidia Tesla S1070.

Figures 12 and 13 depict the runtime breakdown into the main steps of the algorithm for both directions of the transforms in the case of a fixed size of the problem. The shown times are multiplied

by the number of MPI processes. For both transforms, all major operations scale nearly perfectly in the regime studied here. Indeed, as expected and discussed in section 4.1.2, the communication cost grows slowly with the number of processes. As predicted earlier on, section 2.3, the evaluation of the objects, $\Delta_m^{A/S}$, dominates by far over the remaining steps determining the overall run time. In the contrary, the time spent on memory transfers between the host and the GPU devices never exceeds 1 second. In both implementations we assign the evaluation of FFTs to the CPU due to its slightly better performance in comparison to CUFFT also available on Titane (see Figure 14).

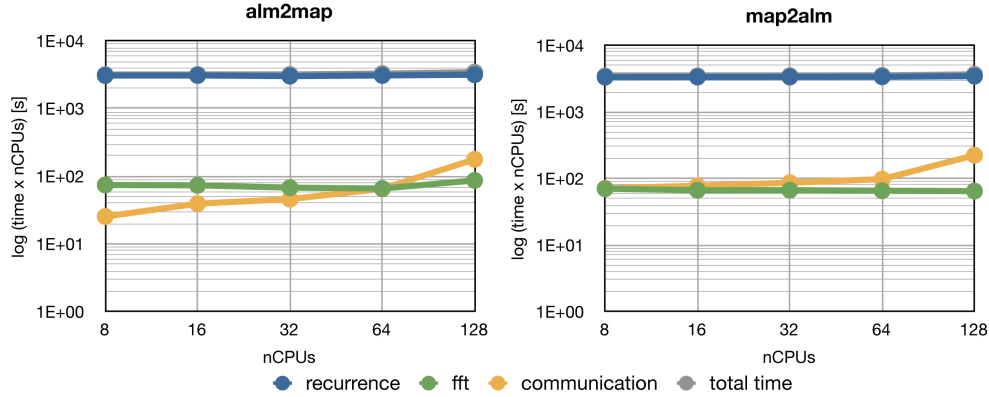


Figure 12. Time breakdown into the main operations for the two transforms, **alm2map_{mt}**, left, and **map2alm_{mt}**, right, in the hybrid MPI/OpenMP case. The results are obtained for experiments with $N_{side} = 16384$ and $\ell_{max} = 2 N_{side}$.

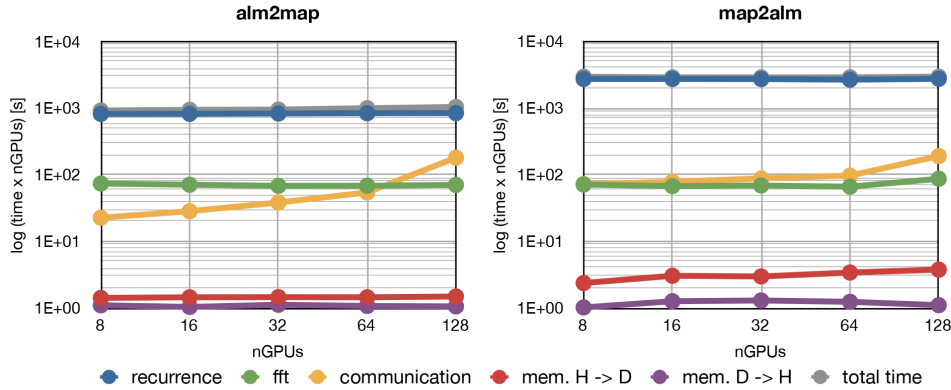


Figure 13. As Figure 12, but for the hybrid MPI/CUDA case.

Finally, in terms of overall runtime, the implementation of **alm2map** for CUDA is, for all our big test cases, on average $\times 3.3$ faster than the multithreaded version. In the contrary, due to the time consuming reduction operations, **map2alm** is only slightly faster than its CPU-based counterpart. An average speed-up of the hybrid implementation MPI/CUDA with respect to the hybrid MPI/OpenMP version is depicted in Figure 15. We emphasise that no SSE intrinsics have been used in the multi-threaded routines used in these tests. In addition, the Nvidia GTX devices used here had double-precision throughput purposefully degraded by the manufacturer to 25% of the full design performance (when compared to Tesla card family with the same chipset).

To assess the accuracy of results on the different architectures used in this paper, we perform a serie of experiments in which we convert the same random set $\alpha_{\ell m}^{init}$ (see beginning of this section)

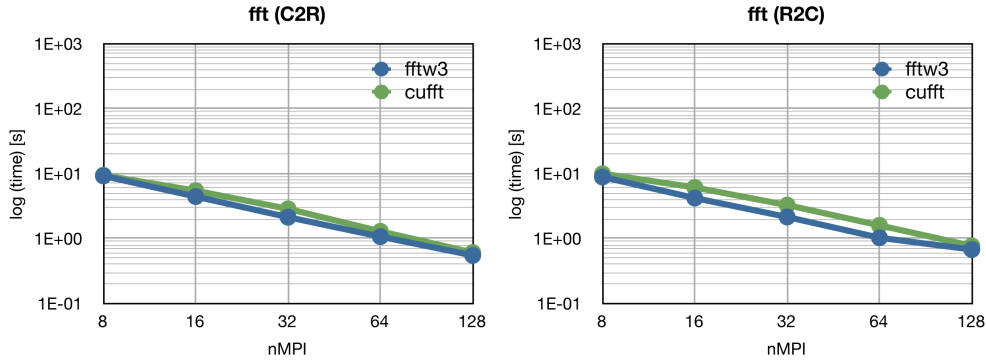


Figure 14. Comparison of two FFT libraries on the Titane supercomputer: CUFFT on GPU and Intel MKL Fourier Transform on CPU. The results correspond to the experiment with $N_{side} = 16384$ and $\ell_{max} = 2 N_{side}$.

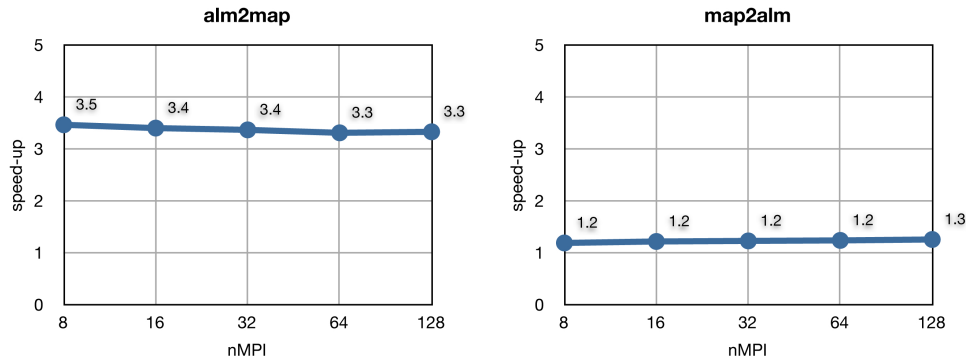


Figure 15. Average speed-up of the hybrid implementation MPI/CUDA with respect to the hybrid MPI/OpenMP version for different number of MPI processes.

to a HEALPIX map on two different architecture, namely Nvidia GPU and x86 CPU. Figure 16 displays the absolute difference of those two maps. In all our tests the maximum absolute difference between two values of the sky signal recovered in the same pixel never exceeded $\sim 10^{-11}$, indicating a very good agreement between these two architectures.

6. CONCLUSION AND FUTURE WORK

This paper describes parallel algorithms for computing spherical harmonic transforms employing two variants of intra-node parallelism, specific to two different architectures i.e., multi-core processors and GPUs, and an optional, distributed memory layer based on MPI directives. We present a detailed discussion of the developed algorithms and perform comprehensive sets of tests comparing them against each other, as well as against an independent implementation of the transforms included in the **libpsht** package. We then evaluate their overall efficiency and scalability. In particular, we show that our inverse SHT run on GeForce 400 Series GPU equipped with latest CUDA architecture outperforms **libpsht**, a state of the art implementation for multi-core processors which benefits from the usage of SSE intrinsics, and executed on the latest Intel Core i7-2600K processor. At the same time the CUDA direct transform, though slower than its inverse-direction counterpart, provides a comparable, though typical lower, performance to that of **libpsht**. For the runs including the MPI layer, we find that a cluster of Nvidia Tesla S1070 can accelerate an overall

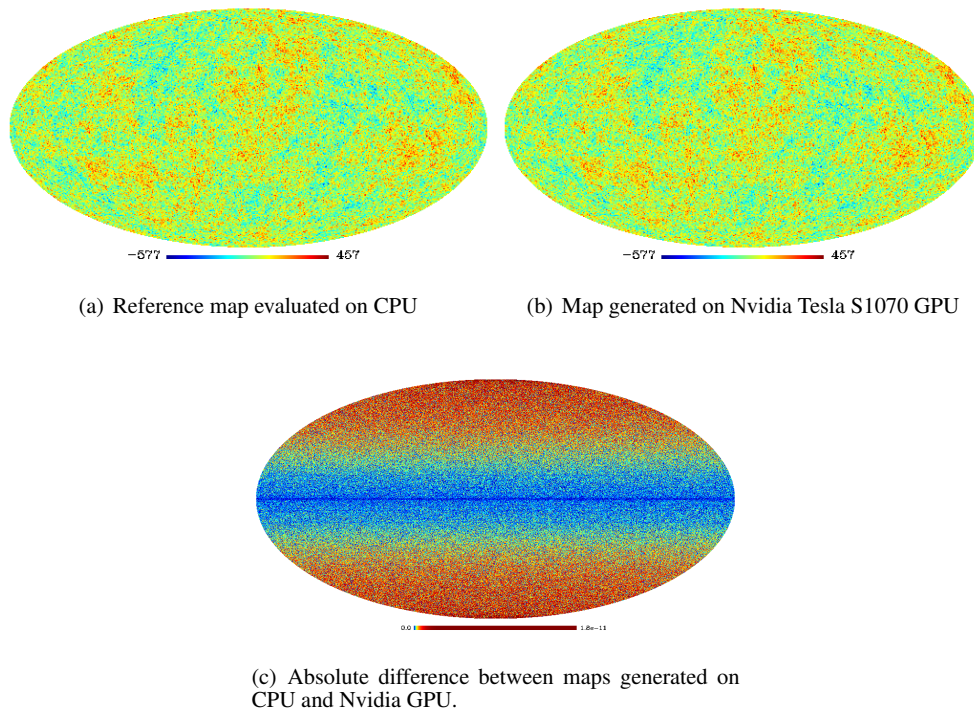


Figure 16. CMB maps generated on two different architectures and their absolute difference.

execution of the inverse transform by as much as 3 times with respect to the hybrid MPI/OpenMP version run on the same number of quad-core processors Intel Nehalem (no SSE intrinsics). The gain for the direct transforms is again found however to be significantly more modest and only about 20 – 30%. We discuss in details the specific features of the nontrivial algorithms invoked for the spherical transforms calculations and elucidate the sources of the inverse – direct transforms dichotomy, which prevent a full adaptation of the latter algorithm to the CUDA architecture.

The current implementation leaves some space for future optimisations. In particular, as a future work, we could add SSE intrinsics (or/and AVX for the latest x86 architectures) to our routines in order to capitalise on a low level vectorization on CPUs. On the algorithmic side and from the application perspective, it would be important to add support of transforms of objects with non-zero spin, which are commonly encountered in the CMB science.

7. ACKNOWLEDGMENTS

This work has been supported in part by French National Research Agency (ANR) through COSINUS program (project MIDAS no. ANR-09-COSI-009). The HPC resources were provided by GENCI- [CCRT] (grants 2011-066647 and 2012-066647) in France and by the NERSC in the US, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Some of the results in this paper have been derived using the HEALPix package [22].

REFERENCES

1. Tanner M. *Tools for statistical inference. Observed data and data augmentation methods*. Springer Verlag, 1992.
2. HEALPIX. URL <http://healpix.jpl.nasa.gov/>.

3. GLESP. URL <http://www.glesp.nbi.dk/>.
4. CCSHT. URL <http://crd.lbl.gov/~cmc/ccshtlib/doc/>.
5. LIBPSHT. URL <http://libpsht.sourceforge.net/>.
6. S²HAT. URL http://www.apc.univ-paris7.fr/APC_CS/Recherche/Adamis/MIDAS09/software/s2hat/s2hat.html.
7. SPHARMONICKIT/S2KIT. URL <http://www.cs.dartmouth.edu/~geelong/sphere/>.
8. SPHERPACK. URL <http://www.cisl.ucar.edu/css/software/spherepack/>.
9. Reinecke M. Libpsht - algorithms for efficient spherical harmonic transforms. *Astronomy and Astrophysics* 2011; **526**.
10. Driscoll JR, Healy DM. Computing fourier transforms and convolutions on the 2-sphere. *Advances in Applied Mathematics* 1994; **15**.
11. Wiaux Y, Jacques L, Vielva P, Vanderghelynst P. Fast Directional Correlation on the Sphere with Steerable Filters. *The Astrophysical Journal* 2006; **652**.
12. Suda R, Takami M. A fast spherical harmonics transform algorithm. *Math. Comput.* 2002; **71**.
13. Tygert M. Fast algorithms for spherical harmonic expansions, ii. *Journal of Computational Physics* 2008; **227**.
14. Mohlenkamp M. A fast transform for spherical harmonics. *Journal of Fourier Analysis and Applications* 1999; **5**.
15. Tygert M. Fast algorithms for spherical harmonic expansions, iii. *Journal of Computational Physics* 2010; **229**.
16. Seljebotn DS. Wavemoth-Fast Spherical Harmonic Transforms by Butterfly Matrix Compression. *Astrophysical Journal Supplement Series* 2012; **199**.
17. Elsner F, Wandelt BD. ARKCoS: artifact-suppressed accelerated radial kernel convolution on the sphere. *Astronomy and Astrophysics* 2011; **532**.
18. Inda M, Bisseling R, Maslen D. On the efficient parallel computation of legendre transforms. *SIAM Journal on Scientific Computing* 2001; **23**.
19. Drake JB, Worley P, D'Azevedo E. Algorithm 888: Spherical harmonic transform algorithms. *ACM Trans. Math. Softw.* 2008; **35**.
20. Soman V. Accelerating spherical harmonic transforms on the nvidia gpu. *Technical Report*, Department of Electrical Engineering, University of Wisconsin 2009.
21. Hupca I, Falcou J, Grigori L, Stomp R. Spherical harmonic transform with gpus. *Euro-Par 2011: Parallel Processing Workshops*, Springer, 2012.
22. Górski KM, Hivon E, Banday AJ, Wandelt BD, Hansen FK, Reinecke M, Bartelmann M. HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. *ApJ* 2005; **622**.
23. Muciaccia PF, Natoli P, Vittorio N. Fast Spherical Harmonic Analysis: A Quick Algorithm for Generating and/or Inverting Full-Sky, High-Resolution Cosmic Microwave Background Anisotropy Maps. *ApJ* 1997; **488**.
24. Swarztrauber PN. *Vectorizing the FFTs, in Parallel Computations*. Academic Press, 1982.
25. Frigo M, Johnson S. The design and implementation of FFTW3. *Proceedings of the IEEE* 2005; **93**.
26. Melton RW, Wills LM. An analysis of the spectral transform operations in climat and weather models. *SIAM J. Sci. Comput.* 2008; **31**.
27. MPICH2. *MPICH – A portable implementation of MPI* 2011. URL <http://www.mcs.anl.gov/research/projects/mpich2/>.
28. Thakur R, Rabenseifner R, Gropp W. Optimization of collective communication operations in mpich. *Int'l Journal of High Performance Computing Applications* 2005; **19**.
29. Bruck J, Ho CT, Kipnis S, Upfal E, Weathersby D. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems* 1997; **8**.
30. Rugina R, Schauser K. Predicting the running times of parallel programs by simulation. *Parallel Processing Symposium, International* 1998; **0**.
31. Sottile M, Chandu V, Bader D. Performance analysis of parallel programs via message-passing graph traversal. *Parallel and Distributed Processing Symposium, International* 2006; **0**.
32. Grigori L, Demmel J, Xiang H. Communication avoiding gaussian elimination. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008.
33. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.
34. Farber R. *CUDA Application Design and Development*. 1st edn., Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2011.
35. Sanders J, Kandrot E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st edn., Addison-Wesley Professional, 2010.
36. Nvidia. *CUDA CUFFT Library* 2010.